TECHNIQUES FOR TRANSPARENT PROGRAM SPECIALIZATION IN DYNAMIC OPTIMIZERS

by

S.Subramanya Sastry

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2003

ACKNOWLEDGMENTS

8 years, 5 houses, 30 housemates, countless officemates is how long it has taken to get this done. I have a number of people to thank who helped me get here, both academically, and personally.

First and foremost, I have to acknowledge the support of my advisors, Jim Smith and Ras Bodik. It has been great working in Jim's group because of the tremendous flexibility and "let-go" atmosphere I worked in. That, more than anything, helped in ways more than Jim himself could possibly know. Coming into Madison, as a newly minted undergraduate, I had a keen interest in computer science and in challenging projects and I had the opportunity to exercise those interests in the Virtual Machines project developing the Strata virtual machine initially and in working on my PhD research problem. That flexibility and easy-going attitude has also enabled me to pursue interests outside Computer Science, sometimes at the cost of my PhD research, which have since profoundly affected and changed me in ways I couldn't have imagined.

On the other hand, it was Ras' push and sustained effort to get me done and graduate which has seen me here (finally!). His role has been wonderfully complementary to Jim's and his effort came in at a time when I needed it most. His relentless stress for clarity in presentation has made the dissertation as good as it is now.

I am also thankful to Mikko Lipasti, Guri Sohi, Charles Fischer, and Susan Horwitz for serving on my prelim and defense committees.

I would also like to thank Mario Wolczko of Sun Labs for his feedback on my work.

Among my numerous officemates, I had the opportunity to work most closely with Timothy Heil. On several occasions, I was able to bounce my ideas off him and get good feedback. In addition, he has made significant contributions to the Strata JVM – he is definitely a better software engineer than I am. Todd Bezenek and Min Zhong have also made critical contributions to the Strata JVM. I am grateful to all the three of them for their contributions. In addition, Manoj Plakal has provided several bug reports that helped in making Strata more usable.

Thanks to Ho-Seop Kim and Quinn Jacobson for doing system administration in the lab in addition to carrying on their research – it can be a thankless job. I've also enjoyed my interactions with all my officemates and several fellow graduate students in Computer Science during all the years I have been here – thanks to all of them for the camaraderie and the fun times.

This work has been supported by NSF grants CCR-9900610, EIA-0071924, and IBM, Microsoft, and Intel.

Outside of academic circles, I have a lot of people to thank – I don't even know how to acknowledge all of them. The person I am today is as much a result of the people that have come into my life and influenced me as it is a result of my own effort. First and foremost, I am thankful and grateful to my parents and my grandmother in ways that words cannot express – they have been tremendously supportive of pretty much anything I have wanted to do and have gone through tremendous hardships to support me and to get me to this stage in my life. Thanks also to my brother and sister for their support and affection. My cousin Amba, and my uncle, Dr.A.B.P.Rao first got me hooked onto computer science. It is doubtful that I would have been so excited by computer science, but for them.

My time at Madison has been made extremely enjoyable thanks to all the people I've lived with – I couldn't possibly name them all. In particular, I have had great times living with Venkatesh Ganti, Vinod John, Harit Modi, and Sridhar Gopal. My final years at Madison at 930 Jenifer St are particularly memorable thanks to the company of Daniel Lipson, Kum Yi Lee, Leigh Rosenberg, Navin Ramankutty, and Jeanine Rhemtulla.

Of all the numerous friends I've made at Madison, I've been strongly influenced by many of my women friends. In particular, Vidhi Parthasarathy has the unique distinction of seeing me through all my changes, my joys, my sorrows, my heartbreaks, and everything else I've been through while at Madison. She has been a wonderful friend, loving and affectionate. In my earlier years at Madison, Muthatha Ramanathan has helped me cope through tough times.

Anita and Leigh have also been profound influences on me and have changed me and my life in

ways I couldn't have predicted. They have, each in their own ways, given me their unconditional support and love and have made my life at Madison all that more beautiful.

Beyond that, I've met a number of wonderful people committed to issues of social justice through my involvement with Asha, AID, Friends of River Narmada, and South Asia Forum. All these volunteers and other social activists have left a deep impression on me and I'm grateful to them for teaching me things about the world that the University setting couldn't have.

And finally, thank you Madison – my times here count for a lot.

ABSTRACT

Program specialization speeds up program execution by eliminating computation that depends on semi-invariant values: known program inputs, or repetitive data encountered during program execution. There are three parts to this problem: (i) determining what values are semi-invariant (ii) determining what code to specialize, and (iii) how to specialize. There is a vast body of literature that addresses the last part of the problem – how to specialize. However, all existing techniques either rely on the programmer or expensive offline program analysis to determine the semi-invariant values, and to determine what code to specialize. Consequently, there is no existing program specialization technique that can be performed at runtime without user intervention (transparently).

This dissertation proposes and investigates techniques for enabling transparent runtime specialization of programs within a dynamic optimizer. The primary contribution of this dissertation is in the form of techniques that address the first two parts above: determining what values are semi-invariant, and determining what code to specialize.

The specializer collects a *store profile* to identify frequently modified portions of the program's data structures. Based on this, the specializer predicts which parts of the program's data structures might be invariant for the rest of the program's execution. This store-profile based invariance detection technique, in addition to being transparent, is more powerful than existing annotation-based approaches. The correctness of this optimistic, but speculative invariance assumptions is ensured by runtime guards that detect modifications of the specialized memory locations. Due to infrastructural limitations, these runtime guarding techniques have not been implemented and studied in this dissertation.

The collected runtime profiles are used by a *scope-building algorithm* that identifies the code regions (specialization scopes) that can be profitably specialized. The identified scopes are specialized using a specializer based on the Sparse Conditional Constant Propagation algorithm that has been extended to eliminate invariant memory accesses as well as unroll and specialize loops. However, other

specializers could be used in combination with the proposed invariance-detection and scope-building techniques.

This dissertation implements a specializer that specializes Java programs available in bytecode form. However, due to infrastructural limitations, the evaluation uses a profile-driven recompilation process (as opposed to true runtime specialization).

Experimental evaluation shows that *transparent runtime specialization is feasible* and can be implemented within dynamic optimizers. The runtime costs are not prohibitive and the specializer can provide good speedups. For example, the specializer could speedup a publicly available Scheme interpreter by a factor of 1.9x. However, with a better engineered implementation than Strata (which is inefficient and unoptimized for runtime compilation), the runtime compilation overheads will be lower leading to to higher speedups. In addition, with an improved specialization model and an improved scope-building algorithm, more specialization opportunities can be exploited which might also translate into higher speedups.

Contents

1 Introduction 1

1.1 Motivations 2

1.1.1 Motivation 1: Performance 2

1.1.2 Motivation 2: Co-designed Virtual Machines 3

1.1.3 Motivation 3: Domain Specific Languages 3

1.2 Example 5

1.3 Implementing a transparent runtime program specializer: Key Challenges 7

1.4 Implementing a transparent runtime program specializer: Key Ideas behind our solution 9

1.5 Summary of contributions 11

1.6 Thesis Organization 13

2 Background and Related Work 14

2.1 Program Specialization 14

2.1.1 Partial Evaluation 14

2.2 Comparing Program Specialization Techniques 16

2.2.1 Exploiting invariance in Runtime Data Structures 18

2.3 Computation Reuse 19

3 Program Specialization in Dynamic Optimizers: An Overview 21

3.1 Specialization model 21

3.1.1 Specialization scopes and keys 21

3.1.2 Specialization transformation 22

3.2 Example 23

3.3 Dynamic Optimizer environment 24

3.4 The specialization process 25
3.4.1 Specialization Meta Issues 26
3.5 Profiling 29
3.5.1 Profile-based Invariance Detection 29
3.5.2 Comparison with Annotation-based Invariance Detection 30
3.5.3 Low-overhead Runtime Profiling 31
3.6 Guarding specialized memory locations 32
3.7 Building scopes 32
3.8 Restricted specialization model: Strengths & Limitations 34

4 Profiling 36

4.1 Profiling for Dynamic Optimizations Systems 36
4.2 Related Work 38
4.3 Proposed Solution 39
4.4 Details of the hybrid profiling scheme 41
4.4.1 Selecting profile information 42
4.4.2 HW-SW communication mechanisms 44
4.5 A framework for designing compressors 45
4.5.1 Samplers as compressors 45
4.5.2 Stratified sampling via hashing 47
4.5.3 Composing profiling components 47
4.5.4 Two-Level Compressors 51
4.5.5 Stratified periodic sampling 52
4.5.6 Hardware cost of the stratified sampler 53
4.6 Reducing Collisions: Adding tags to stratified sampling 54

4.6.1 Design detail 54

4.6.2 Hardware cost 57

4.7 Experimental Results 58

4.7.1 Example Application: Load Value Profiling 58

4.7.2 Methodology 58

4.7.3 Evaluation Metrics 59

4.7.4 Evaluating compressors 61

4.7.5 Sensitivity study of the stratified sampler 64

4.7.6 Two-level compression 66

4.7.7 Simultaneous profiling: Edge and Call target profiling 68

4.8 Summary 69

5 Identifying Specialization Scopes 71

5.1 Outline of algorithm `BuildScopes` 71

5.1.1 Defining an acyclic scope skeleton 73

5.1.2 Building acyclic scope skeletons 74

5.1.3 The skeleton inheritance heuristic 77

5.2 Building scope skeletons for arbitrary SSA graphs 80

5.2.1 Defining a scope skeleton for arbitrary SSA graphs 80

5.2.2 Algorithm `BuildScopeSkeletons` 84

5.3 Identifying scopes from the skeleton set 98

5.4 Identifying a set of non-overlapping scopes 102

6 Creating Specialized Versions 103

6.1 Creating clones of specialization scopes 103

6.2 SCCP-based specialization of the transformed CFG 104

6.2.1 Eliminating invariant memory accesses 106

6.2.2 Specializing Loops 108

6.2.3 Comments on the Loop Specialization algorithm 121

6.3 Related Work 122

7 Implementing lookups: Dispatching to specialized code 124

7.1 Centralized software hashtable 124

7.2 `if-then-else` chain 124

7.3 Scopes as pseudo-methods 126

7.4 Piggybacking onto the method virtual table 128

7.4.1 Chaining of specialized versions 129

7.5 Choosing a lookup implementation 129

7.6 Related Work 130

7.7 Future Work 130

8 Ensuring correctness of specialized code 132

8.1 Implementing Invalidation of Specialized Code 133

8.2 Detecting Invalidation of Specialized Code 134

8.3 Memory Guarding via fine-grained memory protection support 135

8.3.1 Overheads: Qualitative discussion 136

8.4 Software Store Monitoring Schemes 137

8.4.1 Baseline store monitoring 137

8.4.2 Optimization 1: Guaranteeing invariance through program analysis 139

8.4.3 Optimization 2: Reducing the size of the putfield/putarray sets 139

8.4.4 Optimization 3: Integrating store monitors with write barriers 140

8.4.5 Overheads: Qualitative discussion 141

9 Experimental Evaluation **143**

9.1 Research infrastructure and Methodology 143

9.1.1 Details of the Strata compiler 145

9.1.2 Emulating runtime specialization 146

9.1.3 Benchmarks 149

9.2 Evaluating the store and object-access profiles 152

9.2.1 Time and space overheads 153

9.2.2 Accuracy of the store profile 154

9.2.3 Conclusion 158

9.3 Runtime compilation costs 158

9.3.1 Scope-Building Algorithm 159

9.3.2 Specialization Algorithm 160

9.3.3 Summary 162

9.4 Speedup estimates without including specialization costs 163

9.4.1 Caveats 165

9.5 Speedup estimates after including specialization costs 169

10 Conclusions and Future Work **171**

10.1 Summary of contributions 171

10.2 Future Work 174

10.2.1 Improved specialization model 174

10.2.2 Improved scope-building and specialization 176

10.2.3 Systematic study of techniques that detect violations of memory invariance . 177

List of Figures

1.1 Example: Modified version of `FindTreeNode` method in raytrace, a SPECjvm98 benchmark 6

1.2 A Specialized version of `FindTreeNode` 6

3.1 A specialization scope 21

3.2 Pictorial representation of the specialization transformation 22

3.3 Example code region 23

3.4 Outline of the specialization process 26

4.1 Abstract diagram of the stream-compression profiling model. 40

4.2 Hardware profiling components. 48

4.3 Six samplers built from the basic components. 49

4.4 Estimating the frequency of a tuple: Mean error for different input stream lengths. . . 50

4.5 Stratified Sampling Technique 53

4.6 State diagram describing the states of a profile entry 56

4.7 Results for *gcc*: The first graph shows the variation of % error with program progress (up to 4M events). The second graph shows errors for a longer duration (up to 16M events). 61

4.8 Results for *gcc*: The graph shows the variation of cumulative % overhead with increasing time (up to 16M events). The plots for the R_{256} and P_{256} compressors overlap. 62

4.9 Results for all the other benchmarks: The graphs show the variation of % error with program progress (up to 16M events). 65

4.10 Large profiling errors for P_{256} due to periodicity in program behavior – for *db* and *perl* 66

4.11 Sensitivity results of the stratified sampler for four different table sizes for *gcc* and *raytrace* (up to 16M profiling events) 67

4.12 Reduction in overheads due to the A_{16} component 67

4.13 Final error and cumulative overhead for simultaneous call and edge profiling after 4M profiling events 68

5.1 High-level outline of the `BuildScopes` algorithm 72

5.2 `BuildAcyclicScopeSkeletons`: Algorithm to build scope skeletons for acyclic SSA dataflow graphs 75

5.3 Routine `IsInvariantObjectAccessNode` 76

5.4 Example showing two nested skeletons where the outer skeleton subsumes the inner one 78

5.5 Example showing a skeleton in a cyclic SSA graph. The high level source code contains a loop which leads to the two data-flow loops shown in the SSA graph. Since the SSA graph is in canonical form, the constant assignments `i=0`, and `s=0` are folded into the μ-nodes 81

5.6 Example illustrating the utility of condition 5a in the definition a scope skeleton. In this example, the loop cannot be unrolled because the array is defined within the loop. 83

5.7 Example illustrating the utility of condition 5a in the definition of a scope skeleton. In this example, the loop can be unrolled even though the array is defined within the loop. 83

5.8 Conceptual loop transformation that is implemented by the algorithm to collapse dataflow loops that compute a sequence of constants. In the transformed SSA graph, *LC* represents a loop-constant abstract value. 86

5.9 `BuildScopeSkeletons` routine 88

5.10 Example showing the importance of the $\top$-skeleton 91

5.11 ProcessArrayLength, ProcessGetField transfer functions 93
5.12 ProcessGetArray transfer function 94
5.13 BuildNewSkeleton routine 95
5.14 Example illustrating the importance of merging skeletons whose scopes use the same lookup key. The example shows that without merging, there will be unneeded skeleton conflicts. 97
5.15 GetSkeletons routine 98
5.16 Building a scope for a skeleton 100
5.17 Determining number of specialized versions for a scope 101

6.1 Algorithm SCCP_Specialize 104
6.2 The TransformCFG routine 104
6.3 Example showing the power of SCCP as a specializer 105
6.4 Modification to the VisitInstr routine 107
6.5 Example to clarify the interactions between a loop specialization pass and SCCP_Specialize_Basic 110
6.6 Example SCCP visit order 112
6.7 Algorithm to compute a basic block visit order suitable for loop specialization 113
6.8 Enforcing a visit order on SCCP_Specialize_Basic 115
6.9 Algorithm SCCP_Specialize_Loops 117
6.10 State machine used to unroll and specialize loops 119

7.1 Object implementation assumed by the dispatch implementations specific to Java (and potentially other OO-languages) 126
7.2 Implementing a specialization scope as a pseudo-method 127

8.1 Store monitor inserted before the instruction: putfield (value, obj.f) . . 138

8.2 Store monitor inserted before the instruction: `putarray (value, a[i])` . . . 138

8.3 Write barrier used in a concurrent garbage collector by Heil [50] 140

8.4 Integrated store monitor and write barrier 140

9.1 Description of the Strata JVM used to implement the specializer 144

9.2 Passes of the Strata compiler 145

9.3 Interpreter Program 155

9.4 Bubble Sort input program to the interpreter 156

9.5 Original lookup method in `jscheme` 167

9.6 Modified lookup method in `jscheme` and the corresponding specialized code 167

10.1 Factorial assembly program input to the interpreter shown in Figure 9.3 175

10.2 Specialized body of the interpreter loop shown in Figure 9.3 175

List of Tables

4.1 Benchmarks used to evaluate the profiling schemes 59

5.1 Lattice used by the dataflow analysis algorithm that computes scope skeletons on a SSA graph 87

5.2 Transfer functions used by `BuildScopeSkeletons` to build the scope skeletons on the SSA graph 90

5.3 Table showing the values used by the cost-benefit analysis to estimate benefit of a scope 99

9.1 Benchmarks used to evaluate the specialization technique 149

9.2 Specialization parameters (sampling rate and specialization trigger threshold) for the benchmarks 150

9.3 Baseline times for all benchmarks: time to generate Sparc V8 assembly using Strata, time to execute the program with the Strata VM, and time to execute the program with the Hotspot Client VM 151

9.4 Scope-building costs relative to SCCP 159

9.5 Total specialization costs relative to regular compilation 161

9.6 Speedups due to specialization: ratio and absolute times 163

9.7 Consolidating results from Table 9.5 and Table 9.6: the lost column shows speedups when the compilation costs from Table 9.5 are included. 169

CHAPTER 1

INTRODUCTION

Program optimization using traditional static compile-time techniques is becoming increasingly harder because of modular and component-based software development with increasing reliance on general-purpose libraries. In this scenario, dynamic optimization systems [3, 10, 15, 21, 25, 35, 53, 57, 67] are very promising since they can overcome many limitations of static compilation systems as well as offline profile-driven compilation systems.

In this dissertation, our work is targeted at a dynamic optimization systems because they have the following advantages over static and profile-driven recompilation systems:

- *Access to run-time state:* By operating at runtime, dynamic optimization systems have access to a program's runtime state as well as to the fully-linked binary. By leveraging this information, they can build optimization units that are tuned to the runtime behavior of the program. In direct contrast, in the absence of such runtime information, static compilation systems are constrained by the small size of basic compilation units (methods/functions/procedures) as well as hard-to-predict control flow.

- *Transparency:* Unlike profile-driven recompilation systems, dynamic optimization systems tend to be transparent and free the user from the tedious compile/profile/recompile cycle.

- *Adaptability:* Without the need for a separate profiling run, dynamic optimization systems do not suffer from the problem of having to identify a representative input. By monitoring a program's behavior at runtime, dynamic optimization systems can adapt to any change in program behavior by re-optimizing the program as necessary.

While many existing dynamic optimizers contain transparent and adaptive optimization systems [15, 21, 67], they do not fully exploit the advantage of having access to runtime state of a program. They leave unexploited attractive, but hard-to-exploit *data*-specific opportunities due to

stringent time constraints of the run-time setting, and instead focus on the more traditional *control-flow* optimizations developed for profile-guided off-line compilation, like procedure inlining, superblock formation, and code layout.

In this dissertation, we propose to avail this unexploited opportunity by implementing program specialization (a) entirely within a dynamic optimization system, and (b) by exploiting the accessible runtime program state.

Program specialization (in various forms) [8, 31, 47, 56, 58, 61, 65, 71, 73, 75, 76, 80, 81, 86] is a well-studied program optimization technique which speeds up program execution by eliminating computation that depends on constant values. The dynamic optimization environment is arguably the most suitable environment for specializer deployment, because the runtime is where the largest number of constants can be discovered. In addition, adaptability enables better specialization, whereas transparency provides the benefits of complete automation without any programmer intervention.

While there do exist program specializers that operate at run-time, they cannot be implemented in a dynamic optimizer transparently since they require program annotations [31, 47], or else require a separate profiling run and pre-run-time program analysis [71]. This dissertation presents techniques that can enable program specialization to be implemented transparently within a dynamic optimization system.

1.1. Motivations

Having presented the problem tackled by this dissertation, we now present the various reasons why this problem is being tackled, i.e. why is this an important and interesting problem to solve?

1.1.1. Motivation 1: Performance

The first motivation is the performance gains that result from specialization. While specialization is definitely not an optimization that benefits all programs equally, there exist several classes of

programs which benefit from specialization: interpreters, image processing applications, graphics programs, or, in general, any program that operates on semi-invariant data. While offline and staged program specializers exist, this dissertation is concerned with a transparent runtime implementation of a specializer. Transparency has the usual benefits of freeing the user from having to identify specialization opportunities. In certain scenarios, the user might not be able to accurately identify specialization opportunities, especially when the user does not write all the code that goes within an application, for example, when code libraries are commonly used, as in the case of Java APIs.

1.1.2. Motivation 2: Co-designed Virtual Machines

One of the original motivations for this dissertation was to explore the area of co-designed virtual machines [83] along the lines of Transmeta Crusoe [57] and IBM Daisy [40]. This work started as one of many projects to explore different ways in which hardware and software could be designed closely and together to enable better optimizations than is possible without such close h/w-s/w collaboration. Since runtime program specialization relies on low-overhead access to program runtime state (invariant values, frequently executed code regions, data structure access patterns), program specialization is a good candidate for the study of co-designed virtual machines.

1.1.3. Motivation 3: Domain Specific Languages

The next motivation comes from the area of Domain Specific Languages (DSLs) [1]. The area of domain specific languages is an important and growing field of research within the broader research area of programming language design and implementation. DSLs provide semantics customized to the domain of interest which allows programs to be more concise than equivalent programs in general purpose languages (GPL).

An important property of DSLs is that they allow more program properties to be checked than with equivalent GPL programs. This property arises because the semantics of a DSL can be restricted

to enable the decidability of some properties that are critical to a domain [1]. For example, it is well-known that it is impossible to write an algorithm to decide if a program written in a Turing-complete language will terminate on a given input – this is the well-known halting problem. However, by providing looping constructs with restricted semantics, it might be possible to guarantee termination of DSL programs written in the restricted language.

Since it is expensive to build and maintain a compiler, interpreters are the most common way of providing an implementation of a domain specific language. Therefore, interpreters enable fast prototyping and testing of domain specific languages. However, where performance of a DSL program is important, interpreters often fall short. Partial evaluation [56] has been proposed to address this problem. Runtime specializers could be used to specialize DSL interpreters [86] for the input programs. It is well known that specializing an interpreter with respect to an invariant input program produces native code which is equivalent (in principle) to compiling the input program. While realistic specializers might not replace a custom-made JIT compiler for the interpreted language, they may reduce the need to create one by reducing the performance gap.

For example, the DyC project [46–49, 70, 71] was an offshoot of an effort to reduce the extensibility costs [14] of the SPIN extensible operating system [19], which was implemented using a domain-specific language. Similarly, the Sprint project [5] has also employed the Tempo partial evaluator [30–32] to reduce the costs of extensible networks [87]. In this project, the PLAN-P domain-specific language is used to write network protocols which are downloaded into network routers to provide network extensibility. Rather than use an interpreter to implement these protocols, a runtime compiler generated by Tempo is used to speed up execution of these protocols.

To summarize, dynamic transparent specializers might encourage the creation and proliferation of portable DSLs—for example, for safe scripting or for security monitors— because one will have to write only a portable interpreter for the language. From the programmer-productivity standpoint, specializer-equipped execution environments may thus change how certain systems are programmed.

While this approach might not be suitable for all DSL implementations and while our particular specializer does not produce JIT-quality code by specializing interpreters (as most likely, no dynamic specializer will), it is nevertheless a promising approach. Using our approach, we were able to specialize a couple of language interpreters to obtain speedups of 1.7X and 3.6X over unspecialized code. Future work can build on our research to provide even better speedups.

1.2. Example

Let us now look at an example to understand the potential of specialization and the key challenges in automating this process.

Figure 1.1 shows a modified version of the method `FindTreeNode` method of the SPECjvm98 benchmark, *raytrace*. This method compares a `Point` against an `OctNode` which encodes the nodes of an octtree. In this example, the octtree is invariant after it is constructed. Even though the octree has tens of thousands of octnodes, about 60 octnodes account for over 75% of the invocations to `FindTreeNode`. Therefore, if, for each of these 60 octnodes, specialized versions of `FindTreeNode` are created, it can speed up the execution of this method.

Figure 1.2 shows a specialized version of the method for one frequently encountered octnode. Since the octree is invariant, all the six expressions which have chains of five loads are specialized and replaced with constant values. Secondly, the loop is unrolled and all virtual calls to `Find-TreeNode` are replaced with direct calls. Thirdly, since the address of the receivers are known (c_1, ... c_4), if specialized versions of the method exist for any of these receivers, the call target is changed to the specialized version. For this example, specialization is clearly beneficial.

In creating this specialized version of `FindTreeNode`, we needed the following information:

- The octree is invariant
- About 50 frequently seen values of `this` accounted for over 60% of the execution of `Find-`

```
OctNode FindTreeNode(OctNode this, Point p)
{
   if (   p.x <  this.faces[0].verts[0].x
       || p.x >= this.faces[3].verts[0].x)
     return this;
   if (   p.y >  this.faces[1].verts[0].y
       || p.y <= this.faces[4].verts[0].y)
     return this;
   if (   p.z <  this.faces[2].verts[0].z
       || p.z >= this.faces[5].verts[0].z)
     return this;
   if (child[0] != null) {
     for (i = 0; i < 4; i++) {
       found = child[i].FindTreeNode(p);
       if (found != null) return found;
     }
   }
   return null;
}
```

Figure 1.1. Example: Modified version of `FindTreeNode` method in raytrace, a SPECjvm98 benchmark

```
OctNode FindTreeNode_A(Point p)
{
   if ((p.x >  3.5) || (p.x < 4.3) return this;
   if ((p.y > -2.3) || (p.x < -1.0) return this;
   if ((p.z > -0.1) || (p.x < 2.3) return this;
   found = c0.FindTreeNode(p);  /* c0 = child[0] */
   if (found != null) return c0;
   found = c1.FindTreeNode(p);  /* c1 = child[1] */
   if (found != null) return c1;
   found = c2.FindTreeNode(p);  /* c2 = child[2] */
   if (found != null) return c2;
   found = c3.FindTreeNode(p);  /* c3 = child[3] */
   if (found != null) return c3;
   return null;
}
```

Figure 1.2. A Specialized version of `FindTreeNode`

`TreeNode`

- The entire method can be specialized to exploit the invariance of the octree and the repetition of `this`.

Within a dynamic optimizer, all this information has to be discovered automatically which happens to be the key challenge in implementing a transparent program specializer.

1.3. Implementing a transparent runtime program specializer: Key Challenges

In this section, we present the key challenges in implementing a specializer transparently within a dynamic optimizer. In the next section, we present a high-level outline of the specialization technique that addresses them.

In general, any specializer has to find answers to the following questions before it can specialize the input program.

1. *Which variables take on frequently repeating values?* Any code segment using one of these variables could potentially be specialized for each hot value of the variable. In the degenerate case, a variable takes on exactly one value – this variable is then referred to as an *invariant* variable. Profiles are one common way of doing this. However, the challenge is in doing this with low overheads.

2. *Which parts of a data structure are (semi-)invariant?* A memory location is *invariant* if it is *never* modified after it is initialized. A memory location is *semi-invariant* if it is modified one or more times after it is initialized, but exhibits invariant behavior for “sufficiently long” periods of program execution. These invariant and semi-invariant memory locations can be considered run-time constants and used in further optimization.

3. *Which code regions can be beneficially specialized?* These code regions could (but need not)

depend on the answers to the previous two questions. By specializing these code regions, the execution of the program can be sped up.

For example, classic partial evaluation [56] relies on the programmer to annotate a program to answer the first two questions. These answers are then used to specialize the entire program. In other techniques (DyC [47] Tempo [31]), the programmer answers the third question too. More recently, Calpa [70, 71] automated the process of program annotations and can answer all the above three questions automatically. Calpa instruments a program to collect program profiles and analyzes them to generate annotations to answer the above questions and thus automates the process. However, Calpa's technique is not easily transferable to the dynamic optimizer domain because of the high time and space overheads incurred by the initial profiling run (100x-1000x slowdown) and the annotation analysis (2.5x-100x slowdown) [70].

In this dissertation, since we are interested in a transparent implementation, the specializer cannot rely on program annotations and has to discover the answers to all these questions automatically. Equally importantly, these answers have to be discovered with low overheads since the specializer executes at runtime. Thus, the first key challenge facing a transparent program specializer executing at runtime is the low-overhead, automatic identification of repetitive values, semi-invariant memory locations, and specializeable code regions. The primary contributions of this dissertation are in the form of techniques that address this challenge.

The next challenge is in specializing the identified code regions with low overheads and generating low-overhead dispatching code to transfer control to the specialized code. This dissertation makes additional contributions with respect to low-overhead specialization and low-overhead dispatching techniques to transfer control to specialized code.

1.4. Implementing a transparent runtime program specializer: Key Ideas behind our solution

Having examined the motivations for transparently implementing a program specializer within a dynamic optimizer, and the challenges in implementing one, we now present the main ideas that enable such an implementation.

In order to develop a fully automatic specializer, we rely on dynamic program analysis to answer the three questions described in the previous section.

The specializer answers the first two questions with two forms of value profiling: it collects (1) an *object access profile*, which is a frequency distribution of the set of objects accessed at each program point; and (2) *a store profile*, which provides the specializer with the set of memory addresses that were written in the profiled interval. Due to recent advances in *sampling-based profiling* both in software and hardware, these profiles can be collected with high accuracy, yet with overheads that are sufficiently low for dynamic optimizers (5% slowdown and lower) [13, 34, 51, 77, 78, 88].

With the aid of these two value profiles, we turn a Sparse Conditional Constant Propagator (SCCP) [89] into a specializer (based on the online partial evaluation strategy [56]) by letting it consult the two profiles, essentially as follows: given a program point p and an object o that is frequently referenced at p, we perform standard constant propagation starting from p, with two modifications: (1) we assume that o is a constant; and (2) we evaluate load instructions on the concrete memory state (note that the program is specialized while the program is running). The second modification discovers memory invariance: when the specializer encounters a load whose address a is a run-time constant, it consults the store profile. If the memory location a has not been recently written, the specializer assumes that the location is invariant and executes the load to fetch the current content of a. The fetched value is now considered a run-time constant and is constant-propagated further. If the memory location a was recently written, the load is not evaluated (it returns the generic non-constant

value, $\perp$).

However simple, this profile-based specializer-driven identification of memory invariance is both the most novel and most powerful part of the specializer: it enables generation of code that is specialized not only with respect to the hot object *o*, but also with respect to the (temporally) invariant part of the heap. Let us compare our dynamic technique with the alternative of using a static pointer analysis. Since the store profile monitors individual memory locations, invariance detection is more accurate than static pointer analysis working on an abstract heap. It should also be noted that the specializer sidesteps the scalability challenges of pointer analysis in a dynamic optimizer, because it does not rely on any pointer analysis. In contrast to existing specializers, our technique can automatically identify and exploit arbitrary semi-invariance in runtime data structures.

Obviously, the specialization technique is more involved than the above outline may suggest. Firstly, before specialization can be performed, the specializer has to address the third question given above — identifying specializeable code regions. This dissertation shows how a dynamic analysis similar to the one above can be used to identify suitable specializeable code regions. We show that this analysis runs as fast as a standard SCCP algorithm making it quite efficient. Secondly, for more powerful specialization, the SCCP-based specializer is extended to unroll loops and specialize them. Finally, the dynamic analysis must (sometimes) be guarded with checks protecting against writes into run-time-constant memory locations. In this dissertation, we propose two techniques for implementing these guards. The first is based on a recently proposed fine-grained programmable memory protection scheme [90]. The second technique relies on the type safety of Java to insert runtime guards at all potential modification sites.

Our specializer crucially relies on runtime profiles for a low-overhead, accurate, automatic identification of repetitive values, invariant memory, and specializeable code regions. Such profiling is possible due to recent advances in *sampling-based profiling* both in software and hardware [13, 34, 51, 77, 78, 88]. Of these existing proposals, one of them [77] is proposed and evaluated as part of this

dissertation. This dissertation proposes general-purpose profiling support for use in dynamic optimizers. This profiler can collect a wide range of profiles including store and object access profiles needed by our specializer. This profiler is a h/w-s/w hybrid: it relies on simple hardware support for accumulating profiling events and sending this information to software which processes it to build the desired profiles. The profiling hardware implements stratified sampling – which is a sampling technique in which the input population is divided into disjoint sub-populations and sampled independently [43]. The profiler stratifies the input event stream with a hashing function. In our experiments, for the same desired accuracy and profiling time, stratified sampling incurs half the overheads of h/w random sampling.

1.5. Summary of contributions

Having presented the motivations that underlie this dissertation and the key ideas behind our specialization technique, we now summarize the contributions of this dissertation.

This dissertation proposes the following techniques and shows how they enable a transparent implementation of runtime program specialization.

- We show that profile-guided invariance detection techniques can automatically identify invariance in runtime data structures at fine granularities.
- We show that using profile-guided dynamic analysis, specializeable code regions can be identified automatically with low overheads.
- We show that the necessary value profiles can be collected accurately and with low overheads using a general-purpose profiling technique suitable for dynamic optimizers. More specifically, we show that very simple stratified sampling hardware enables this profiling technique.

These techniques have been used to implement a transparent program specializer. We have evaluated this specializer for specializing Java bytecodes within a Java Virtual Machine [63]. We sum-

marize the practical advantages of this specializer below:

- *It can be performed entirely at run-time, without any user intervention or off-line preprocessing.* Our algorithms automatically specialize unmodified Java bytecode, avoiding the need for programmer-inserted annotations required by other specializers (except for Calpa [71], which however works pre-runtime). With hardware support similar to existing hardware or proposed hardware [57, 90], these techniques could also be adapted for programs written in C.

- *It offers accurate invariance detection.* Our dynamic analysis detects invariance at fine granularities unlike existing specialization approaches which are limited in their ability to exploit partially static data structures. Our analysis detects invariance of individual, concrete heap locations, not that of abstract ones. Guarded with run-time checks (either software or hardware), the analysis can exploit specialization opportunities that may not be easily detectable or annotatable on the (abstract) source code, in particular: (i) temporally semi-invariant data structures, such as infrequently-updated hash tables, and (ii) spatially semi-invariant data structures, such as a linked list containing both variant and invariant elements.

- *It is sufficiently lightweight for run-time compilation.* We show that the algorithm that identifies specializeable code regions is as efficient as a Wegman-Zadeck sparse conditional constant propagator (SCCP) [89]. Further, we show that we can adapt the SCCP algorithm to eliminate invariant memory, unroll loops, and create multiple specialized versions of the same code region. Our reliance on the readily available SCCP makes our techniques easy to implement. Despite its transformational simplicity, it can speedup some programs by up to a factor of 3.6X, and simple extensions (future work) can provide further significant improvements.

The dissertation also implements and evaluates the h/w-s/w hybrid profiling scheme that implements the stratified sampling scheme in hardware. While this profiling scheme was developed to enable runtime specialization, it is a stand-alone contribution of this dissertation and can be used out-

side the context of our specific specialization system. Within the profiling domain, the contributions of this dissertation can be summarized as follows:

- We propose a h/w-s/w hybrid profiling model which is based on hardware preprocessing of a stream of profiling events.

- We present a framework of abstract profiling components which can be composed together to construct and evaluate different profilers. In this dissertation, we study six different profiling schemes.

- Specifically, we propose *stratified sampling* as the profiler of choice. Our evaluation shows that with fixed profiling overheads and accuracy levels, this sampler achieves the desired accuracy at least twice as fast as a random sampler.

1.6. Thesis Organization

In Chapter 2, we present a background for this dissertation and discuss work related to this dissertation. As we present the details of our technique in later chapters, we will present work related to that specific component being discussed. In Chapter 3, we present an overview of our specialization technique. In Chapters 5 to 8, we present details of the various components of our technique. In Chapter 9, we present an experimental evaluation of our technique and conclude with a summary and future work in Chapter 10.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we first present an overview of the area of program specialization. We then discuss the phenomenon of *computation reuse* which helps us situate program specialization within a broader context.

2.1. Program Specialization

Program specialization (in various forms) [8,14,30–32,47,56,58,61,65,71,73,75,80,81,86] is a well-studied program optimization technique. In essence, it is a technique that speeds up program execution by eliminating computation that depends on constant values.

In the following sections, we present an overview of some of these specialization techniques. First, we present a discussion of partial evaluation, which is at the heart of most specialization techniques. Next, we present a comparison of several existing specializers.

2.1.1. Partial Evaluation

Partial Evaluation is perhaps one of the most well-studied program specialization technique.[1] In its classic form, partial evaluation [56] is a compile-time source-to-source optimization technique applied to the entire program. It is used to speed up execution of a program $\mathtt{P}$ with input $\mathtt{I} = \{\mathtt{S}, \mathtt{D}\}$ where $\mathtt{S}$ is the subset of the input which is known *a priori*, and $\mathtt{D}$ is the subset of the input which is not known *a priori*. Given $\mathtt{S}$, a partial evaluator specializes $\mathtt{P}$ for $\mathtt{S}$ and yields a new program $\mathtt{P}_\mathtt{S}$. When $\mathtt{P}_\mathtt{S}$ is then provided the rest of the inputs $\mathtt{D}$, it generates the same result as $\mathtt{P}$ would generate for the input $\mathtt{I} = \{\mathtt{S}, \mathtt{D}\}$, but presumably much faster. Thus, the essence of applying partial evaluation is

[1] Neil Jones *et al.* [56] defines partial evaluation and program specialization to be equivalent. However, Mogensen [72] defines partial evaluation to be an instance of program specialization. Mogensen lists the technique of writing generating extensions as an example of non-partial-evaluation based program specializers. In this dissertation, we use program specialization to refer to the concept of specializing a program (rather than any specific technique), whereas partial evaluation refers to a very specific specialization technique.

to use some relatively static (unchanging) input `S` and generate faster special-purpose programs $\texttt{P}_\texttt{S}$. Note that while partial evaluation is typically applied to an entire program, it can easily be applied to smaller scopes such as procedures.

Many techniques for implementing partial evaluation have been developed. They can be classified into two categories: *offline* partial evaluation and *online* partial evaluation. These terms do not refer to the times when the technique is applied (compile-time vs. run-time), rather either technique can be performed at compile-time, or at run-time, or can be staged.

Offline partial evaluation is a two-stage process. In the first stage, a binding-time analysis (BTA) uses the knowledge of the static input `S` and assigns binding times (static/dynamic) to all program variables and annotates all program statements as being *reducible* (can be evaluated at partial evaluation time) or *residual* (has to be evaluated at run-time). During this analysis, the actual values of the static inputs are never used. The only information used is the knowledge of what variables are static. The second stage, known as the specialization phase, uses the values of the static variables and eliminates all statements marked reducible and retains all statements marked residual. DyC [47], Tempo [31], CMix [8], and Fabius [61] are some examples of the offline approach. While CMix, DyC, and Tempo target the C programming language, Fabius targets the functional language ML. Because of their design, offline partial evaluation techniques are perfectly suited for a staged compilation model. The expensive BTA can be performed at compile-time without the knowledge of the actual runtime values. The specialization phase (which in this case simply follows the BTA annotations) can be performed at run-time when the actual runtime values become available. DyC [46] is a good example of this approach. Tempo [31] can be performed entirely at compile-time or can be staged like DyC.

In contrast with offline partial evaluation techniques, online partial evaluation techniques seem more suitable for a runtime optimization environment since they make specialization decisions using the knowledge of invariant values. An online partial evaluation technique [56, 76] is a single-stage

process in which the values of the static variables are used to make specialization decisions. Online partial evaluation techniques can potentially discover more specialization opportunities when compared to offline techniques because they have access to the runtime program state (including memory state) which can be used to make better specialization decisions. The primary drawback of online partial evaluation techniques is the high specialization overheads that will be incurred at runtime. In addition, online partial evaluators tend to be more aggressive in duplicating code and have to address problems of termination. FUSE [76], a partial evaluator for a subset of the functional language Scheme, is an example of an online partial evaluator. FUSE is a compile-time online partial evaluator.

2.2. Comparing Program Specialization Techniques

Having presented a brief overview of partial evaluation which forms the basis for many specialization techniques, we now discuss some of them in relation to how they answer the specialization questions listed in Section 1.3. Recall that all specializers have to essentially find answers to three questions before specializing the program:

- Identifying repetitive inputs: what program variables exhibit repetition?
- Identifying semi-invariant memory: what parts of runtime data structures are (semi-)invariant?
- Identifying specializeable code regions: what code regions can be specialized?

In this section, we present a comparison of several existing specialization techniques with respect to how they answer these questions.

In order to specialize a program, it is important to identify the variables that exhibit repetition (if the variable takes on only one value, it is called an invariant variable). Specialization techniques use these repetitive variables to seed the specialization process.

For specializers of functional languages, repetitive variables are specified by user annotations or by using function currying [61]. In an annotation-based approach, the inputs to a function are annotated as being *static* (invariant) or *dynamic* (Example: Mix [56]). The specializer uses these annotations to specialize this function (and potentially other functions invoked by it) for the values of the static variables. In the function currying approach, a function that requires multiple inputs is only provided some of the inputs. The interpreter then returns a new function that takes the rest of the inputs. Currying can be implemented by returning wrapper functions. However, program specializers can return a more optimized function in which the computation dependent on the partial input is specialized away.

CMix [8], Tempo [31], and DyC [47] – specializers for the C programming language – require the user to provide a specification of both the input variables to specialize for as well as the code regions to specialize. Recently, Calpa [70, 71] automated the process of generating annotations for DyC by using runtime profiles collected with an instrumented executable. Calpa successfully generates annotations that were hand-generated for DyC. However, Calpa requires a separate profiling run and suffers from drawbacks similar to those of profile-driven recompilation systems. There might not be any representative input available that captures the inputs during the non-profiling runs. One approach to transparent runtime specialization would be to adapt Calpa for use within a dynamic optimization system. However, the techniques used by Calpa to automate the annotation process does not appear to be suitable for this purpose. Calpa requires extensive profiles of use and definition of scalar variables, as well as profiles of loads of and stores to pointer variables. The cost of collecting these profiles is very high. The profiling run incurs a slowdown from 170X to 2000X [70]. Thus, while Calpa is well-suited as a tool to automate the annotation process for DyC, its high overheads makes it hard to adapt it for a runtime implementation.

Our specializer relies on runtime profiles to identify variables exhibiting repetition and to identify semi-invariant parts of data structures. Section 1.4 presented the profile-based technique that enables

automatic identification of semi-invariant parts of runtime data structures. In addition, an automatic scope-finding algorithm identifies the code regions that can be profitably specialized. Our technique differs from most existing specialization techniques since it does not rely on any user annotations. It is similar in spirit to Calpa in automating the specialization process. However, unlike Calpa, our technique can be used to implement runtime specialization without any compile-time program analysis. This is enabled by a combination of (i) low-overhead runtime profiling techniques, (ii) restricted specialization model, and (iii) a greedy, but fast and efficient scope-building techniques. We present an overview of our technique in Chapter 3 and present further details in the later chapters.

2.2.1. Exploiting invariance in Runtime Data Structures

In this section, we compare specialization techniques with respect to their support for exploiting invariance in runtime data structures. Programs written in functional languages are side-effect free. Consequently, data structure values are no different from primitive values. Specializers for functional languages can exploit partially invariant data structures. However, computation via side-effects is one of the defining characteristics of imperative languages like C and Java. As a result, data structure values (pointers in C, object references in Java) cannot be treated the same way as primitive values. Repetition of a pointer variable is not sufficient for using the contents of the locations referenced by the pointer variable. The underlying referenced data structure must also be invariant. Consequently, program specializers for C (CMix, DyC, Tempo) either rely on whole-program alias analysis techniques [8, 31] or rely on programmer annotations to detect invariance of runtime data structures [47].

Our specializer differs from these existing techniques in its ability to automatically detect invariance in runtime data structures unlike existing specialization techniques. In Section 1.4, we described how the specializer uses the store profile to accomplish this. In addition, unlike existing annotation-based techniques, the store profile enables exploitation of fine-grained irregular invariance in data

structures (Section 3.5.2).

2.3. Computation Reuse

In this section, we situate program specialization within the more general phenomenon of *computation reuse*.

Most programs contain computation regions which compute the same results that they computed previously, regardless of how one defines a computation region (individual machine instructions, basic blocks, dependence chains, expressions, dynamic execution traces, functions, methods, or any arbitrary code segment). Reusable computations present an optimization opportunity since the results of repetitive computation can be stored and reused without having to compute the results again. As an example, loop-invariant code motion [6] computes the value of a loop-invariant expression once outside the loop, caches the result in a register, and it within the loop. Common subexpression elimination (CSE), and partial redundancy elimination (PRE) are other examples of static compiler optimizations that are also computation reuse techniques. As another example, classic memoization [69] is an optimization in which the computation of entire functions is reused by caching the results in a memo-table.

In general, computation reuse opportunities can arise either due to invariance in the input data or because of program structure (the way programs are written). Loop-invariant code motion (LICM), CSE, PRE are examples of reuse opportunities that arise due to program structure.

Program specialization is a computation reuse technique that eliminates reusable computation that arises due to invariance in input data. The reusable computation regions need not be contiguous – the specializeable code can be mixed in with non-specializeable code. These techniques optimize a program by hard-coding the results of the reusable computation region within the program. When compared to other computation reuse techniques (like memoization, instruction reuse, basic-block reuse), program specialization is perhaps one of the more powerful computation reuse techniques

because it can exploit reusable computation at far coarser granularities than these other techniques. For the same reason however, they are quite general-purpose and can be expensive to apply at fine granularities.

Another computation reuse technique closely related to program specialization is memoization. One of the defining characteristics of memoization is that the reusable computation region is always contiguous, i.e. the region does not contain any non-reusable instructions. Memoization is the underlying idea behind a number of specific techniques: standard memoization [18, 20, 28, 69], Instruction Reuse [84], Polymorphic Inline Caching for virtual calls in object-oriented languages [53], compiler-directed reuse proposed by Connors and Hwu [29], Basic-Block reuse by Huang and Lilja [55], and Data Specialization [58]. These memoization techniques differ in what is memoized, the key for looking up the reuse table, and how the reuse table is implemented. Classic memoization [18, 20, 28, 69] reuses computation at the granularity of functions, data specialization [58] reuses computation at the granularity of expressions, Instruction Reuse [84] reuses computation at the granularity of individual instructions, and so on.

CHAPTER 3

PROGRAM SPECIALIZATION IN DYNAMIC OPTIMIZERS: AN OVERVIEW

In Chapter 1, we sketched the main idea of our specialization technique. In this chapter, we present a more detailed overview of our specialization technique using an example.

3.1. Specialization model

We first describe the specialization model that is the basis of our specialization technique. There are three components to our specialization model:

- *Specialization Scopes* which are the atomic units of specialization and correspond to control-flow regions in the program,
- *Specialization Keys* which correspond to input variables of a specialization scope that exhibit repetition, and the
- *Specialization Transformation* which specifies a program transformation based on selected specialization scopes and keys.

3.1.1. Specialization scopes and keys

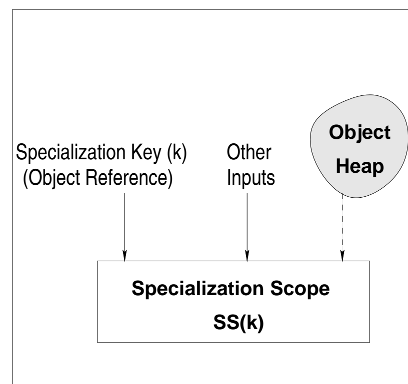

Figure 3.1. A specialization scope

The unit of specialization is a single-entry, multiple-exit region of a control-flow graph (CFG), called a *specialization scope*, or simply *scope*. A scope is entirely intra-procedural and does not cross method boundaries, but may contain method calls. A scope may also contain entire loop nests. A scope may have two kinds of inputs: explicit, consisting of object reference and scalar variables, and implicit, consisting of the heap (see Figure 3.1). Of the variables making up the explicit input, one object reference is selected to be the specialization *key*. A scope is specialized with respect to this specialization key: the specializer creates multiple specialized versions of the scope, one for each frequent value of the key. In the rest of this dissertation, we use `SS(k)` to denote a scope `SS` with a key `k`. Also, for purposes of brevity, we will occasionally refer to `SS(k)` as a scope, even though we mean *scope* `SS` *with a key* `k`.

We now translate some of these concepts into partial evaluation terminology. The key corresponds to the static division of the explicit inputs of the scope. Creating multiple versions of the same scope corresponds to polyvariant specialization (one for each value of `k`). Some partial evaluators also support polyvariant divisions – a partial evaluator that implements polyvariant divisions specializes the same code region with different keys (i.e. different `k`'s). However, our model does not support polyvariant divisions because given a scope, the key is fixed.

3.1.2. Specialization transformation

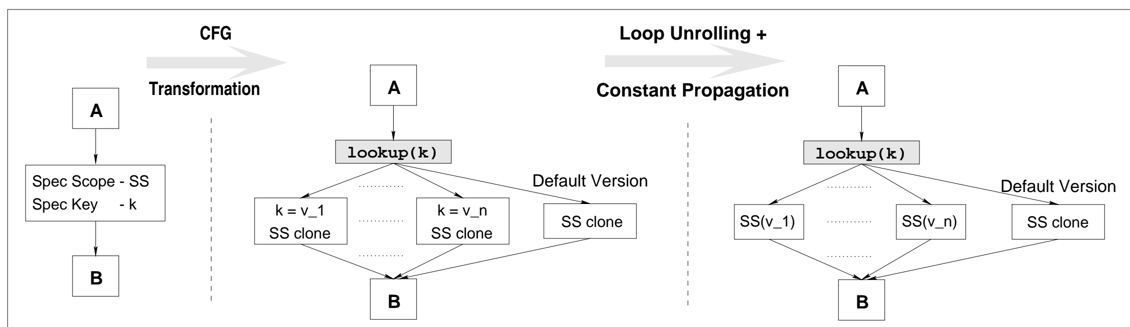

Figure 3.2. Pictorial representation of the specialization transformation

Figure 3.2 shows the two-step process of specializing a scope. First, the scope is cloned for each hot value v_i of the key `k`. Second, each cloned scope is specialized under its hot value. Third, a new instruction, `lookup(k)`, is introduced to transfer the control to the appropriate specialized version depending on the value of the key. A failed lookup will transfer control to the original scope.

3.2. Example

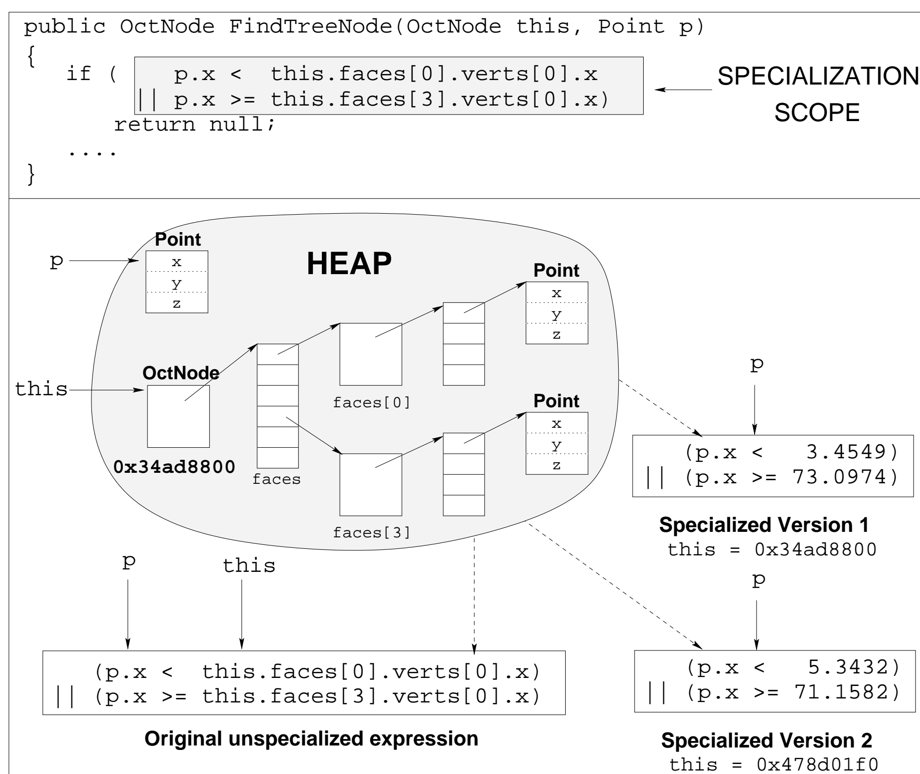

Figure 3.3. Example code region

Having presented the specialization model, we introduce an example which we use as a running example in the rest of this section to illustrate the workings of our specialization system.

Figure 3.3 shows a fragment of the method `FindTreeNode` from the SpecJVM98 benchmark *raytrace*. This method compares a `Point` against an `OctNode`, encoded as a relatively complex pointer-based data structure (see Figure 3.3), which the method traverses whenever it evaluates an

expression like `this.faces[0].verts[0].x`. Altogether, five memory loads are needed in this expression, an inefficiency caused in large part by Java semantics.

Since the octtree is invariant after it is constructed, this method can be specialized for frequently visited nodes of the tree (those near the root). Let us assume that the specializer has identified the specialization scope highlighted in the figure. The key variable is `this`. For this example, let us consider a and b as the two frequently occurring values of the key. Note that the argument `p` is not invariant and thus belongs to the dynamic scope input. Figure 3.3 shows the scope specialized for `this=`a and `this=`b. For simplicity, we do not show the `lookup(this)` statement. Note that all loads from invariant memory have been specialized away.

In this chapter, we will use this example to show how our specializer:

1. identifies the invariant portions of the octtree;

2. determines a beneficial scope, highlighted in the figure; and

3. creates the specialized versions of the scope.

Before we present the steps of our specialization process, we present the dynamic optimizer environment within which it is implemented.

3.3. Dynamic Optimizer environment

Our specializer is designed to run in an adaptive dynamic optimizer like Hotspot [67] or Jikes RVM [4] (aka Jalapeno). These optimizers execute continuously on separate threads of control concurrently with the application. We assume that a dedicated thread collects the value profiles needed by our specializer. We assume that specialization is triggered at fixed time intervals, or whenever the count of profiled events reaches a certain threshold. This triggering could be done by a dynamic controller that is part of adaptive compilation systems like HotSpot and Jikes. The length of these profiling/specialization phases can be determined either empirically or by using some tuning

algorithms that adapts to the working set of the program. In this dissertation, the length of these phases have been determined empirically and are fixed for the entire execution of the application.

Our specializer faces the same issues as other dynamic optimizers, namely fast and efficient code generation, identification of "hot" methods, on-stack replacement of optimized code [53], cost-benefit analysis, and synchronization between the main thread and the optimizer threads. These problems are not unique to our specialization technique. Existing solutions apply in our setting [11, 53, 67] and we don't specifically address them in this dissertation. For example, low-overhead dynamic optimization techniques, such as [11, 12, 60, 67], reduce overhead by means of multiple threads of control that share the processor with the optimized application. Krintz *et al.* [59] show that by using compiler-generated annotations as part of Java bytecode classfiles, dynamic compilation overheads can be reduced. Such techniques have broad applicability in a dynamic optimization environment and can help reduce the optimization overheads of our specialization system too.

3.4. The specialization process

In Figure 3.4, we show the steps performed by our specializer when it is invoked. First, for each hot method identified by the optimizer, specialization scopes are identified by an analysis that relies on a dynamic analysis similar to that used by the SCCP-based specialization algorithm we discussed in Section 1.4. We present the details of this algorithm in Chapter 5. This algorithm relies on two value profiles: the object access profile and the store profile. In this dissertation, we propose a general-purpose profiling solution for collecting a range of profiles at runtime with low-overhead. We discuss the details of this profiling solution in Chapter 4.

After the specialization scopes are identified, the CFG is transformed as shown in Figure 3.2 and its scopes are specialized, as discussed in Section 1.4. The detailed algorithm is presented in Chapter 6. During this specialization, the addresses of invariant memory locations participating in specialization are collected. If the absence of modifications to these *specialized memory locations*

```
while (true) {
    Wait for a signal from the optimization controller;
    Let hm  = Set of hot methods (obtained from the optimization controller);
    Let oap = Object Access Profile;   /* global variable */
    Let sp  = Store Profile;           /* global variable */
    Let iml = {};    /* Set of specialized memory locations */
    for each (m ∈ hm) {
        SS = BuildScopes(m);                          /* Chapter 5 */
        im = SCCP_Specialize(m, SS);                  /* Chapter 6 */
        Add memory locations in im to iml;
        Generate lookup implementations;              /* Chapter 7 */
    }
    Add guards for memory locations in iml;           /* Chapter 8 */
    Chain together specialized versions;              /* Section 7.4.1 */
    Reset sp;
    Decay profile counters in oap by half;
}
```

Figure 3.4. Outline of the specialization process

cannot be proven at specialization time using type-safety, run-time guards are added to detect such modifications. When a modification is detected, the particular specialized scope is immediately invalidated (see Chapter 8). Next, for each scope, a suitable implementation of the lookup statement is generated (see Chapter 7). Finally, a subset of lookup instructions are eliminated by "chaining" specialized scopes together with direct calls.

3.4.1. Specialization Meta Issues

We now discuss some specialization meta issues: first, when and does the specializer system become active, and second, how does the specializer deal with scenarios of no-performance-benefit from specialization, and significant-performance-benefit from specialization. These issues deal with how a runtime specializer fits in within the overall dynamic optimization system. These issues are specific to the dynamic optimization setting and are not pertinent for an offline specialization system.

Specializer startup

We first discuss how and when a specializer becomes active. Clearly, the specialization system can be activated right at the beginning of the application's execution. However, this has the disadvantage that the specializer might discover specialization opportunities that are short-lived and correspond to the initialization phase. Therefore, there needs to be a mechanism for skipping over the initialization phase. There are multiple ways of doing this.

One straightforward approach would be to set a fixed empirical time threshold after which the specialization system is triggered. However, besides being ad hoc, it is not a robust technique.

Another approach would be to monitor the activity of the class loader and native compiler. When a program first starts executing, a dynamic optimization JVM (like HotSpot or Jalapeno) loads the necessary Java classes, compiles them to native code and also optimizes them (either right away or based on feedback from runtime profiles). During the initialization phase, the class loader will be very active as it loads up all the required classes. In addition, the compiler in a dynamic optimizer like Jalapeno will perform an initial compile of the active methods and will also be very active in the initialization phases. Thus, another approach would be to wait for the class loader activity and the baseline optimizer activity to reach a low watermark before triggering the specialization system.

Having seen specializer startup, let us now examine steady state behavior of the specializer. Very broadly, there are two scenarios: (i) where specialization provides no benefit, and (ii) where specialization provides benefit.

Handling the case of non-zero specialization benefit

Let us first consider the steady-state behavior for programs which benefit from runtime specialization. This scenario closely follows the specialization process described in Section 3.4. After the specializer starts up, the specializer is triggered periodically. At that time, it analyzes the profiles, builds specialization scopes, and generates specialized code as described in Section 3.4.

Handling the case of no runtime specialization benefit

Let us now consider the case when specialization is not beneficial. Note that specialization is not a general purpose technique and might not lead to performance improvement for all programs. Therefore, the specialization technique has to have small overheads on such programs. We now describe techniques to accomplish this.

The specialization process in Figure 3.4 shows that profiling and scope building precede the actual specialization transformation. These costs are incurred by all applications – irrespective of whether specialization is beneficial or not. We now discuss ways of keeping these overheads small to minimize slowing down programs that won't benefit from specialization.

As regards profiling, our evaluation shows that the overheads of collecting these profiles using a hardware stratified sampler will be about 5-10% of the program's execution time. Using a software profiling technique like that proposed by Arnold and Ryder [13], the costs are expected to be under 10%, but, with lower sampling rates than supported by hardware profiler. While these costs are small, these costs are non-negligible. However, when the specializer determines that specialization is not beneficial, profiling can be turned off (or alternatively, the sampling rate can be reduced). This approach lowers the profiling cost and might keep it small. This is similar to Dynamo's behavior; dynamic optimization is turned off when Dynamo determines during the early part of the program execution that the expected benefits might not offset the cost of dynamic optimization [15].

As regards identification of specialization scopes, evaluation of the specializer shows that building specialization scopes is a low-cost step. Even with an inefficient implementation of the specializer studied in this dissertation, the average cost of identifying specialization scopes for a hot method was under 150 ms on an UltraSparc workstation – this cost is likely to be smaller in a well-tuned dynamic optimizer. In addition, if specialization is performed by a separate optimization thread, the cost will be further reduced. If the specializer does not identify any specialization scopes, no additional transformation cost is incurred.

A simple way of lowering the cost of scope identification for non-beneficial applications would be to progressively increase the length of the profiling phases (either linearly, or exponentially). As a result, the specializer waits longer and longer before trying to build new specialization scopes. For programs where no scopes are identified, this back-off technique has the effect of lowering the cost of scope building significantly.

Having presented the big picture, we now present an outline of the different components of our solution. We first examine the profiling component enables the automatic identification of invariant values and automatic identification of specialization scopes.

3.5. Profiling

The *object access profile* accumulates the frequency distribution of $(i, \textit{object-address})$ pairs, where *i* is any instruction of type `getfield`, `getarray`, and `arraylength`. The value *object-address* is the actual address of the object referenced by i. Note that in each specialization phase, this profile is decayed by half, giving preference to specialization of recently accessed objects.

3.5.1. Profile-based Invariance Detection

The *store profile* accumulates addresses of recently stored-to memory locations. As mentioned above, any memory location absent in the store profile is considered invariant.[1] Note that in each specialization phase, the store profile is reset (see Figure 3.4). This is done to detect temporally semi-invariant data. Due to this resetting, all store-addresses show up in the store profile once (if at all[2]) and disappear after the next reset. Therefore, all previous modifications to a memory location are forgotten enabling the specializer to exploit invariance during the periods when the content of the memory location is unchanged.

[1]However, one could also imagine implementing a more probabilistic notion of invariance where the store frequency is used to determine the likelihood of the location being invariant.

[2]A store-address may not show up in the profile at all due to sampling errors

So, all store addresses that correspond to data structure initialization disappear after they show up the first time. Looking at the example in Figure 3.3, since the octree is invariant after it is constructed, the store addresses corresponding to this construction will appear only in the initial phases. When the scope and specialization algorithms query the profile during later phases, they will assume the invariance of the memory locations in the octree.

Thus, our profiling technique enables us to identify arbitrary spatial and temporal invariance in data structures – the store profile enables this spatially, the resetting enables this temporally.

3.5.2. Comparison with Annotation-based Invariance Detection

Let us compare the power of profile-based invariance detection with that of static annotations, whether produced by a programmer or by a tool.

First, let us consider *class-based* annotations, where the annotation states that some fields of all objects of a particular class are run-time invariant. This annotation would fail to specialize Figure 3.3. In this example, we want to state, among other properties, that the fields `x, y, z` in all `Point` objects in the octtree are invariant; unfortunately, there are `Point` objects outside the tree whose fields are not invariant, for example the object passed via argument `p`, and so a class-based annotation would be incorrect in this case.

Next, let us consider *expression-based* annotations, which identify static expressions that always access invariant memory locations. In Figure 3.3, we could annotate `this.faces[0].vert[0]` as invariant. However, if a part of the octtree were modified,[3] then this annotation would be incorrect. To conclude, although more fine-grain static annotations have been proposed runtime store-profile based schemes are a simple and powerful way to detect invariance.

[3]in reality, the octtree is not modified in the raytrace benchmark

3.5.3. Low-overhead Runtime Profiling

We now present an overview of our profiling scheme used to collect the above value profiles.

In this dissertation, one of our goals is to provide profiling support for dynamic optimizations. Ideally, a profiler suitable for dynamic optimization should exhibit the following properties: (1) have low overheads (2) have high accuracy, (3) converge rapidly, and (4) be broadly applicable to enable collection of a variety of profiles. We propose a hybrid profiling model which leverages the strengths of both hardware and software. Programmable hardware monitors events of interest, collects data at high bandwidth, does some minimal preprocessing and passes on the information to software. Software (virtual machine software, dynamic optimization runtime systems, etc.) collects this information, analyzes it as necessary and builds the necessary profiles (edge profiles, basic block profiles, etc.).

On the hardware side, we propose a family of hardware profilers based on a model of *stream compression*. The processor generates a stream of (selected) profiling events which are then compressed (in that the bandwidth is reduced) by a hardware preprocessor before sending the compressed message stream to software. We show that hardware that implements *stratified sampling* [43] is a good enough compressor. Stratified sampling divides (stratifies) an input population into multiple disjoint subpopulations and samples them independently. It is a well-known statistical result that this process is more accurate than ordinary random sampling. In our proposed solution, we use a hashing scheme to stratify the input event stream. We show that a stratified sampler achieves a desired accuracy twice as fast as a random profiler while incurring the same time overheads.

We present the full details of our profiling technique and present the results of our evaluation in Chapter 4. While this profiling technique has been developed to enable our runtime specialization, it can be used in other applications.

It should be noted that this hardware-based profiler is not a precondition for our transparent specializer. Other profiling software or hardware profiling solutions [13] can be used in place of the

profiling technique developed in this dissertation.

3.6. Guarding specialized memory locations

It should be clear that the store profile merely shows that a memory location is unlikely to be modified in the future. Consequently, our store-profile based specialization is a speculative optimization and we need to provide mechanisms to guarantee the correctness of this speculation. In this dissertation, we show how the specialized memory locations can be guarded to detect modifications. When such modifications are detected, the corresponding specialized versions are invalidated. We propose two techniques for guarding memory locations. The first technique employs fine-grain programmable memory protection, such as Mondrian [90]. A less powerful version of such memory protection is also available in Transmeta Crusoe processors [57]. This technique requires programming the memory system to raise an interrupt after a store to any of the specialized memory locations. It works for specialization of unsafe languages like C where efficient pointer analysis is very hard. The second technique relies on the type safety of Java. We first examine the specialized locations as object fields; if these fields are all final, or no putfields to these fields outside constructors exist, these locations can never be overwritten. Otherwise, we instrument putfields to check if they are overwriting a guarded location. These mechanisms are discussed in detail in Chapter 8.

3.7. Building scopes

Having presented an overview of most of the specialization system, it remains to outline how our specializer determines beneficial specialization scopes. Creating a scope essentially boils down to identifying a suitable specialization key variable and a suitable key lookup point (this is where specialization based on the key will start). The choice of the two is extremely important for specialization quality. Consider a loop that traverses an invariant linked list. Ideally, we would like to completely unroll and specialize away the entire loop. Clearly, a suitable key is the pointer that

walks through the list. Regarding the lookup point for this key, a good choice is outside the loop, as it will induce a scope that includes the entire loop; a bad choice for the lookup is inside the loop, as this lookup placement will create a scope that includes only one iteration—a much less powerful specialization because a lookup is invoked on each iteration and the loop is also not specialized away.

The goal is thus to find suitable key variables and build the largest possible specialization scopes for these key variables (with the constraint that excessively large scopes cause too much code duplication). Let us now look at the main idea behind the algorithm.

Main Idea: For each program point `p` that accesses an object `o`, the algorithm computes a scope starting at `p` and having `o` as the key. To do this, the algorithm essentially tries to construct a skeleton of instructions (rooted at `p` and connected by data-flow dependencies) that access invariant memory locations and identifies a single-entry, multiple-exit control-flow boundary that encloses this skeleton. The algorithm only focuses on the subset of instructions involved in accessing objects: `getfield`, `getarray`, and `arraylength` Java bytecode instructions.

The algorithm starts a new specialization scope whenever the process of growing a skeleton is disrupted: due to a variant object access, due to an intervening call, and other reasons which are precisely described in Chapter 5.

This process also determines which loops should be unrolled. The detailed algorithm is presented in Chapter 5 formulated as a dataflow analysis. Note that just like the specialization algorithm, the scope-building algorithm uses the value profiles to determine invariant memory accesses. It is also similar to binding-time analysis used by offline partial evaluators in that it is a pre-pass that determines what to specialize.

Example: Let us now walk through the steps involved in identifying the scope for the example shown in Figure 3.3. The algorithm starts at the first getfield instruction `p.x` and retrieves the hot object referenced at this site. On consulting the store profile, it finds that the field `p.x` is modified for

this object. So, the algorithm proceeds to the next getfield instruction `this.faces`. By consulting the object-access profile, the hot object is selected as a representative object for further exploration. Let us assume it to be `a = 0x34ad8800` as shown in the figure. The algorithm discovers that `a.faces` is invariant by consulting the store profile. So, a new scope is started at this getfield instruction. Using this instruction as the skeleton, the algorithm proceeds to the dataflow successor: `getarray(faces[0])`. By accessing the object-access and store profiles, the algorithm discovers that this is an invariant access and grows the skeleton. Continuing in this fashion, the scope is greedily grown to include both the expressions highlighted in Figure 3.3.

3.8. Restricted specialization model: Strengths & Limitations

In our current specialization model, we have imposed two restrictions on specialization keys. Firstly, we disallow scalars from being specialization keys. Secondly, we only allow single-element specialization keys. These restrictions are both our strength and our limitation. They help us in the following ways:

- The no-scalar restriction reduces the profiling overheads by only requiring us to profile object accesses which are far fewer in number than scalar accesses. This restriction also allows us to implement low-overhead lookup mechanisms to transfer control to the right specialized version. This will become clear in Chapter 7 when we present several lookup implementations.
- The single-element key restriction simplifies our automatic scope-building algorithm by eliminating the need to analyze the interactions of multiple invariant variables. This restriction also reduces the profiling overheads by eliminating the need to profile multi-element tuples.

However, the restrictions limit us in the following ways:

- The single-element key restriction limits us by not allowing us to exploit specialization opportunities that require a multi-variable key. A good example is the `equals` Java method which

is implemented by many classes. These methods essentially compare two objects for equality. Currently, we cannot exploit these opportunities.

- The no-scalar key restriction limits us by not allowing us to exploit specialization opportunities that do not arise from object references. For example, "pure" mathematical functions that only operate on scalar values cannot be exploited by our approach.

In practice, this simple model appears to be adequate enough to exploit enough specialization opportunities to make this approach beneficial. Addressing these limitations to enable better specialization is left for future work. This work is a first step in implementing transparent program specializers in dynamic optimization systems. The goal is to get good performance, not necessarily the best possible performance. Therefore, we made these above simplifications to enable us to study simple and efficient specialization techniques. Future work can build on this work and lift these restrictions. Lifting the no-scalar key restriction is easier than lifting the single-element key restriction. With multi-element keys, there will be a cross-product of choices that need to be analyzed at every two-input instruction. This affects both the profiling technique as well as the scope-building algorithm. Developing efficient techniques to handle this potential blow-up in choices is the key challenge in lifting this restriction.

Having presented an overview of the entire specialization process and the big picture, we are now ready to present the individual components in greater detail in the following chapters. In Chapter 4, we present a general-purpose profiling technique for dynamic optimizers. In Chapter 5, we present the details of the scope building algorithm. In Chapter 6, we present the details of our specialization algorithm. In Chapter 7, we discuss implementations of the lookup instruction, and in Chapter 8, we present the guarding mechanisms needed to detect modifications to memory assumed to be invariant.

CHAPTER 4
PROFILING

In this chapter, we present a profiler capable of collecting a variety of profiles at runtime with low overheads and high accuracy. This profiler has been designed to collect profiles rapidly for use within a dynamic optimization system. This profiler is a stand-alone contribution of this dissertation, independent of the specializer, and can be used in a dynamic optimization system for collecting profiles for various runtime optimizations. Consequently, this chapter is presented from the perspective of this broader context. Where applicable, we will present a discussion of the particular needs of the specializer studied in this dissertation.

4.1. Profiling for Dynamic Optimizations Systems

The specialization technique proposed in this dissertation relies on runtime profiles to automatically identify specialization opportunities. Like most optimizations implemented in dynamic optimizers, it is a profile-driven optimization. Profile-driven optimizations (also known as feedback-directed optimizations) refer to program optimizations that improve program performance by analyzing a program's runtime profile to discover information static compile-time analysis typically cannot infer. In general, runtime profiles have become indispensable in a spectrum of advanced optimizations, both within the static compile-time optimization domain as well as the dynamic run-time optimization domain. These optimizations include trace scheduling [41] and extend well beyond it: basic-block, edge and path profiles [17, 91] identify hot spots in the program; call-graph profiles [7] guide procedure inlining [11,23,24]; dynamic-type profiling removes indirect calls in object-oriented languages [53, 54]; value-invariance profiles lead to program specialization [22, 29, 71, 73]; and memory-conflict profiles allow aggressive load-store reordering [39,42], and others [10, 26, 40, 57].

While low-overhead, rapid profiling support can benefit both static as well as dynamic optimizers, such support is more critical for dynamic optimizers. In this dissertation, we are primarily

interested in providing general-purpose profiling support for dynamic optimizers. Ideally, a profiler suitable for dynamic optimization should have the following properties:

- *It should collect profiles with low overheads.* Because profiling takes place during execution, its overhead must be significantly smaller than the optimization benefit (which depends on the specific runtime optimization). Optimizations like specialization *can* provide speedups beyond 2X whereas a more typical benefit from program optimizations is about 10% or less (Ex: optimizations implemented in Jikes RVM [12]). This severely constrains the tolerable profiling overhead for a *general-purpose* program profiler. Even with a more aggressive optimization like specialization, low profiling overheads enable more fine-grained specialization than is possible otherwise.

- *The collected profiles must have high accuracy.* The accuracy of the profiler impacts the quality of optimizations. For example, if a memory location that is written does not show up in the store profile, the specializer might incorrectly specialize it away and the specialized version will get invalidated at a later time. In order to minimize invalidations, high accuracy of the store profile is critical. In general, while the accuracy requirements vary across profiling applications, the higher the accuracy of the profiler, the faster the profile converges to within an acceptable error. Rapid profiling, in turn, leads to earlier optimization and correspondingly longer execution in the optimized mode. In Dynamo [38], rapid selection of hot paths was important for maximizing returns from dynamic optimizations.

- *It should have broad applicability.* The diversity of dynamic optimizations calls for a versatile profiler that can measure diverse properties of control flow, addresses, and data values.

- *It should support simultaneous profiling.* Sometimes it is convenient to collect multiple profiles simultaneously. For example, the specializer developed in this dissertation relies on object-access and store profiles which are needed together to enable runtime specialization.

Consequently, it is important to be able to collect these profiles simultaneously. As another example, if the dynamic behavior of an optimized procedure changes, the optimizer may want to trigger its re-optimization. If multiple profiles can be collected simultaneously, monitoring of changes can run on the background of continuous optimizations.

- *It should have low cost and complexity.* Minimal hardware support and simple software algorithms bring the well-known benefits of reduced power consumption and verifiability.

4.2. Related Work

Let us review related work with respect to the above ideal properties, focusing on three distinct implementation categories: smart software profilers, custom hardware profilers, and hybrid profilers.

Smart software profilers: The first group of software profilers instruments the program with profiling instructions. One method for reducing the overhead of executing the additional instructions is to exploit the program structure: Ball-Larus edge profiling [16] and path profiling [17] use program analysis and manage to restrict overheads to 10-30%. Other tricks for reducing the instrumentation overhead include restricting profiling to a subset of instructions [22,73] and turning off profiling after the profile stabilizes [22]. Despite recent advances, profiles that measure more than the control flow incur high overheads. For example, the best software value profiler slows down the program 10–30 times [22, 71].

The second software approach is sampling. Burrows *et al.* proposed sampling techniques to collect value profiles with low overhead (about 10%) [66]. However, the low sampling rate (1 every 32,000 instructions) increases the time before optimizations can be performed. More recently, Arnold and Ryder [13] proposed a low-overhead instrumentation-based sampling technique for collecting profiles within a Java Virtual Machine. Their results show within the Jalapeno JVM, with a

sampling rate of 1000, they can collect two different profiles simultaneously with high accuracies (99%) and low overheads (3%-6%). Hirzel and Chilimbi [51] extended this instrumentation technique and adapted it for use within Vulcan, a x86 binary rewriting system [85]. Using this technique, they show that they can collect temporal profiles of data streams with low overheads (3%-18%).

Custom hardware support: Conte *et al.* proposed a *profile buffer* [33] and Merten *et al.* described a *hotspot detector* [68]. While these specialized designs work very well for their specialized purpose, they cannot be used to collect other kinds of profiles.

Hybrid profilers: In these solutions, programmable hardware collects profiling information, potentially performs some simple profile preprocessing, and passes on the information to software which then does a more complete profile analysis. ProfileMe [36], the Relational Profiling Architecture (RPA) [88], and the programmable profiling co-processor [34] are examples of this approach. ProfileMe [36] provides mechanisms of instruction-based profiling wherein the hardware picks instructions and collects a variety of information as instructions flow down the pipeline. The information is post-processed by software. The other two solutions (RPA and the profiling co-processor) provide more flexible profiling abilities than ProfileMe, by supporting a wider range of profiles to be collected in hardware and by enabling hardware preprocessing of profile information that reduces post-profile analysis overheads in software.

4.3. Proposed Solution

We explore a method that combines the advantages of hardware and software profiling. While hardware is suitable for high-bandwidth data processing, software is a better fit for irregular processing of small amounts of data. We choose a hybrid hardware-software approach because purely hardware approaches are usually inflexible and are targeted at specific optimizations. Purely soft-

ware approaches tend to incur relatively large overheads and/or require low sampling rates which can lead to long profiling times.

Our solution is motivated by the observation that most kinds of profiles compute *execution counts* of certain events of interest (values computed by an instruction, targets of a branch, targets of a call, or addresses of a load). This observation leads to a *stream compression* profiling model. In this model, the processor generates a stream of profiled events whose type is selected by software. The stream is a sequence of data tuples: for example, a value-profiling tuple contains the PC of a load instruction and the loaded value. Dedicated hardware compresses this stream before passing it on to software. The basic compression method collapses and counts identical tuples. Consequently, profiling overhead is reduced because software processes a shorter stream.

A conceptual view of the hybrid stream compression profiling model is shown in Figure 4.1. The *selector* creates a tuple for each retired instruction chosen for profiling and sends the tuple to the *compressor*. The compressor, placed off the processor's critical path, consumes the selected tuple stream. The compressor summarizes the input stream and feeds it to profiling software through an intermediate buffer. In Figure 4.1, the input tuple stream $t_1, t_2, t_1, t_2, t_3, t_2$ may be compressed into an output tuple stream $(t_1, 2), (t_2, 3), (t_3, 1)$. The second element in the output pairs denotes the number of occurrences of the tuple.

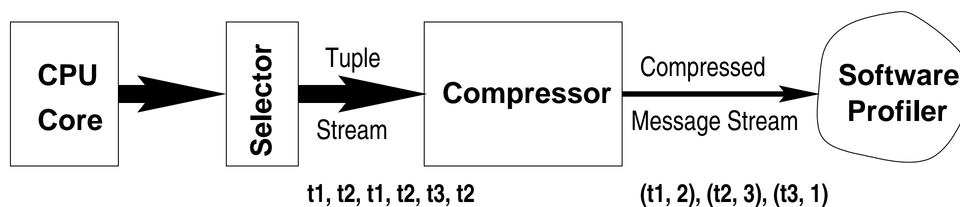

Figure 4.1. Abstract diagram of the stream-compression profiling model.

The variety of profiles that can be collected by the stream-compression model depends on the flexibility of the software-controlled selector. In this study, we evaluate our profiler on value profiles of load instructions [22], a very demanding profiling application. Besides that, we also show that

the hybrid profiling scheme can collect multiple profiles simultaneously. As an application, we show that edge profiles can be collected simultaneously with call target profiles with high accuracy and low overhead.

In this dissertation, we primarily focus on designing "efficient" compressors that incur low overhead by means of high compression of the tuple stream, while preserving adequate profile accuracy. We present a set of profiling components that can be composed in a number of ways to yield different compressors. Because optimizations can tolerate some profiling errors, we allow the stream compressor to be lossy, thereby enabling a low-cost sampling-based hardware design. Conventional simple random sampling, simple periodic sampling, stratified random sampling, and stratified periodic sampling are among the compressors considered. *Stratified sampling* is a technique in which the population is split into multiple disjoint sub-populations (strata) which are then sampled independently [43]. In our implementation, we use a hash function to split the input stream. Using load value profiling as a case study, we experimentally show that given a fixed overhead budget and a desired level of accuracy, the stratified periodic sampler achieves the desired accuracy the fastest.

Having presented the broad contours of our profiling technique, we present the details of this technique in the following sections. In Section 4.4, we present some details of the selector shown in Figure 4.1. In Section 4.5, we show a variety of sampling compressors and discuss their properties using Monte Carlo simulations. In Section 4.6, we describe a more sophisticated, tagged compressor that serves as a reference point in our experiments. We present the results of our evaluation in Section 4.7.

4.4. Details of the hybrid profiling scheme

There are three main components of the hybrid profiling model: the selection mechanism, the compression mechanism, and the communication of information to software. In this section, we discuss the first and third of these components. Compression mechanisms are central to our profiling

solution and are discussed in greater detail in the following section.

4.4.1. Selecting profile information

Referring to Figure 4.1, the “front-end” of the profiler is a selection mechanism. Although specific selector implementations are not our main focus, one possibility is programmable selector hardware. In particular, registers in the selector can be written by profiling software via special instructions. These registers can then be used by hardware to select the specific instructions to be profiled. Heil and Smith [88] describe such mechanisms as part of their relational profiling architecture. Below, we describe a few selection mechanisms.

- **Selection by opcode**: A number of profiling applications can be implemented by selecting instructions on the basis of opcode. This strategy is a low-cost solution that does not require any modification to the ISA or the program binary. This mechanism is proposed in [88] and [34].

- **Selection by PC range**: This strategy is useful for focusing on hot segments of a program or for reducing the profile bandwidth to the hardware profiler. This mechanism was proposed in [34].

- **Binary modification**: This mechanism places explicit instructions in the binary immediately before an instruction to be profiled. This mechanism requires one extra opcode in the ISA and also introduces overheads in the instruction stream. However, this method can be used to supplement the above two selection methods because it enables profiling sets of instructions that are not otherwise easily selected.

For collecting the store and object-access profiles needed by the specializer, we simply select instructions for profiling based on the opcode. For the value profiling application we use to evaluate the profiler, opcode-based selection is sufficient. For most of the commonly used profiles, opcode-based

selection seems to be sufficient. Furthermore, in general, while the profiling tuples can be arbitrarily long, two-word profile tuples seem to be sufficient for most profiling applications.

Since the compressor does not interpret the contents of profile tuples, the same hardware can be used without modification for collecting a variety of profiles as listed below.

- Store Profile: the tuple is `<0, memory-address>`.
- Object-Access Profile: the tuple is `<load-PC, memory-address>`.
- Load Value Profile: the tuple is `<load-PC, load-value>`.
- Value Profile: the tuple is `<PC, instr-output>`.
- Edge Profile: the tuple is `<branch-PC, branch-target>`.
- Call Target Profile: the tuple is `<call-PC, call-target>`.
- Type Profile: the tuple is `<call-PC, method-table-addr>`; this profile enables feedback-directed inlining [11] and polymorphic inline caching [54].

Since the compressor does not interpret the tuple, it is possible to run multiple profiling applications simultaneously. The software is responsible for distinguishing between the incoming tuple messages. When profiles being collected can be distinguished solely on the basis of opcode, the software can distinguish between incoming messages on the basis of the message PC. For example, if an edge profile and call target profile are being collected simultaneously, they can be distinguished on the basis of the PC: the former profiles branches and indirect jumps and the latter profiles profiles direct and indirect calls. Likewise, for our specializer application, when we collect the store profile and object-access profiles simultaneously, we can distinguish them simply on the basis of the PC. For the store profile, the PC is always 0, whereas for the object-access profile, it is non-zero. However, this technique for distinguishing between profiles does not always work. For example, if edge profiles and a profile of mispredicted branches are to be collected simultaneously, this approach

cannot be used because the same branches might be present in both profiles. One possible approach is to add tags to profiling events belonging to different profiling application. For example, the selector can assign mispredicted branches with a different tag than correctly predicted branches. These tags enable the software in separating messages belonging to the two profiles and construct them accurately. The tags could also be used by the compressor to enable better compression by keeping the streams separate. These tagging mechanisms have not been studied in this dissertation and are presented here for the sake of completeness.

4.4.2. HW-SW communication mechanisms

There are two mechanisms for communicating tuple messages to software. The first mechanism is based on processor interrupts and the second is based on message passing.

Interrupt-based approach.

In this approach, tuple information is stored in a hardware buffer. When the buffer fills, the main processor is interrupted. The interrupt handler reads the buffer and folds the tuple information into the profile data being collected. This approach is used in [9, 34].

Message-passing based approach.

In this approach, profile information is communicated to concurrent software threads via shared queues in memory. These concurrent threads read profile messages from the shared queues and compute the profile. An example of this approach are the *service threads* outlined in [88]. Service threads running on simple service processors read and process the messages. The relevant overhead metric for this approach is the time between consecutive messages sent to the service threads. The time should be long enough so that the profile service thread can completely process the message.

In our experimental evaluation, we assume a interrupt-based communication model because it is

readily implementable in current generation processors.

4.5. A framework for designing compressors

This section focuses on the lossy stream compressor, which is the most novel part of our hybrid profiling scheme. Instead of presenting a few independent compressor designs, we describe a framework of components from which various compressors can be built.

4.5.1. Samplers as compressors

Samplers can serve as stream compressors because selecting a subset of the input stream reduces the bandwidth of the output stream. Samplers count the input events statistically: an event t selected by a sampler operating at a rate r can be interpreted by software as r occurrences of tuple t compressed into one. Clearly, such stream compression is *lossy*, because events skipped by the sampler may have been different than t. However, when the stream is *biased* towards tuple t (i.e., the stream is dominated by t), the sampler's accuracy may be sufficient for profiling purposes. This subsection presents two basic samplers used in our framework, and the following subsection focuses on increasing the bias in the input stream.

- A *random sampler* with rate r, denoted R_r, selects an element of the input stream with probability $p = 1/r$. Note that random samplers in [9, 36, 66] are slightly different; they select an element via a countdown register initialized with a random number from the interval $\langle 1, 2r \rangle$.
- A *periodic sampler* with rate r, denoted P_r, selects every rth element of the input stream. In statistical literature, this sampler is known as a *systematic* sampler [43].

In order to design an accurate compressor, the inherent accuracy of the two samplers must be understood. In this section, we assume an idealized input stream in which tuples are randomly permuted

(the reason for this assumption is that an input stream with periodic behavior may cause large errors with the periodic sampler P_r). In Section 4.7, we will evaluate the compressors on real workloads.

We compare R_r and P_r using a very simple profiling problem. Assume the *output* stream of the compressor contains k elements. The problem is to determine how many elements were in the input stream. Clearly, the answer for both R_r and P_r is rk. More precisely, rk is the *most likely* number of elements seen in the input. The important difference between the two samplers is *how* likely it is that their input actually contained rk elements. The P_r sampler is more confident about its answer because it effectively counts the input stream; its input stream length is **guaranteed** to contain between rk and $rk + (r - 1)$ elements. On the other hand, the length of the input stream for R_r can range from k to infinitely many elements. Therefore, R_r **estimates** the lengths of the input stream. This explanation is validated experimentally. The error of the two samplers is shown in Figure 4.4 using an experiment described in detail in Section 4.5.3.

We can address the drawback of R_r by adding to it a counter that measures the length of the input stream. We use C to denote a counter component of our framework, and CR_r to denote a random sampler equipped with such a counter. With this enhancement, CR_r has access to the same information as P and the two are thus equivalent. We later show that CR_r is equivalent to P_r in terms of profiling accuracy. The extra information provided by C can be used to more accurately estimate the desired input parameter when compared to R_r.

One problem of a periodic sampler (P_r) is that the sampling period might synchronize with periodic behavior in the input stream (for example, those generated by programs) which can lead to greater inaccuracies. Figure 4.10 shows examples of inaccuracies due to such periodicity that is present in profiling input streams generated by programs.

4.5.2. Stratified sampling via hashing

As mentioned in the previous subsection, the accuracy of sampling increases with the bias in the input stream: the more a tuple dominates the stream, the less likely is the sampler to make a mistake. An effective technique for increasing the bias is to stratify the input population into disjoint sub-populations which are then independently sampled. This technique, known as *stratified sampling* [43], can be conveniently implemented in the profiling context using hashing. Because the input stream is split based on tuples having the same hash signature, it can be reasoned that any given substream has a greater bias (or, smaller entropy, in information theoretic terms; lower variance, in statistical terms) than the original input stream. Hence, the samples that are selected from each substream are more accurate representatives of the input stream than a corresponding same-size sample selected from the original input stream.

We use $H[X_r]_n$ to denote a *hash-based splitter* that splits the input stream into n disjoint substreams using a hash function. Each of the substreams is independently sampled at the rate of r using any sampler X. We discuss the stratified sampler in more detail in the following subsection. Its implementation details are described in Section 4.5.5.

4.5.3. Composing profiling components

Figure 4.2 pictorially shows the four profiling components R_r, P_r, C, and $H[X]_n$ that we presented in the previous subsections.

The input to each of these components is the input stream. The random and periodic samplers produce on the output a sample of the input stream. The hash-based splitter produces multiple disjoint input streams. The counter produces the length of the input stream, i.e. the number of input tuples. We now show how these components can be composed to produce different sampling schemes.

Figure 4.3 shows six different samplers created by combining the four basic components: P_r,

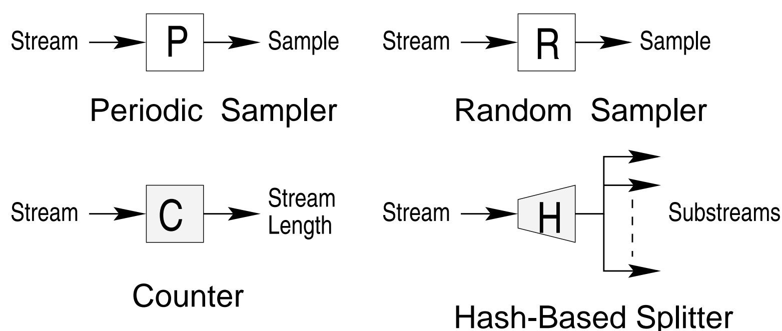

Figure 4.2. Hardware profiling components.

R_r, CR_r, $H[P_r]_n$, $H[R_r]_n$, and $H[CR_r]_n$. All samplers compress the input stream into a stream of messages $\langle tuple, count\rangle$. For the samplers with the counter component C, the value $count$ equals the count since the last sample was taken. For samplers without the counter component, the value $count$ is implied and equals the sampling rate r.

We will show that the *stratified periodic sampler* $H[P_r]_n$ sampler is the most successful design. In statistical literature, such a sampler is called stratified systematic sampler [43]. We evaluate the accuracy of these samplers using the problem of estimating the number of times, t, that a given tuple p occurs in the input stream of length N (note that this is a slightly harder problem than the one introduced in Section 4.5.1). We evaluate the samplers using Monte Carlo simulations. We generated the input stream by randomly permuting a sequence of N tuples containing t copies of the tuple p ($t = 0.3N$). The randomly generated stream is input to each sampler and the output stream is used to estimate the value of t, as follows. For each sampler without a counter, if the output stream contains i copies of tuple p, then t is estimated to be $EST = ir$. For each sampler with a counter, t is estimated to be the sum of $count$'s from all messages that contain the tuple p. We plot the error in the estimate $ERR = |100(t - EST)/EST|$.

Figure 4.4 shows the error in estimating the frequency of the tuple for stream lengths ranging from 1000 to 20000, for a sampling rate of 10. For each input stream length N, we performed 2500 experiments and plotted the mean value of ERR. As expected, the graph shows that with increasing

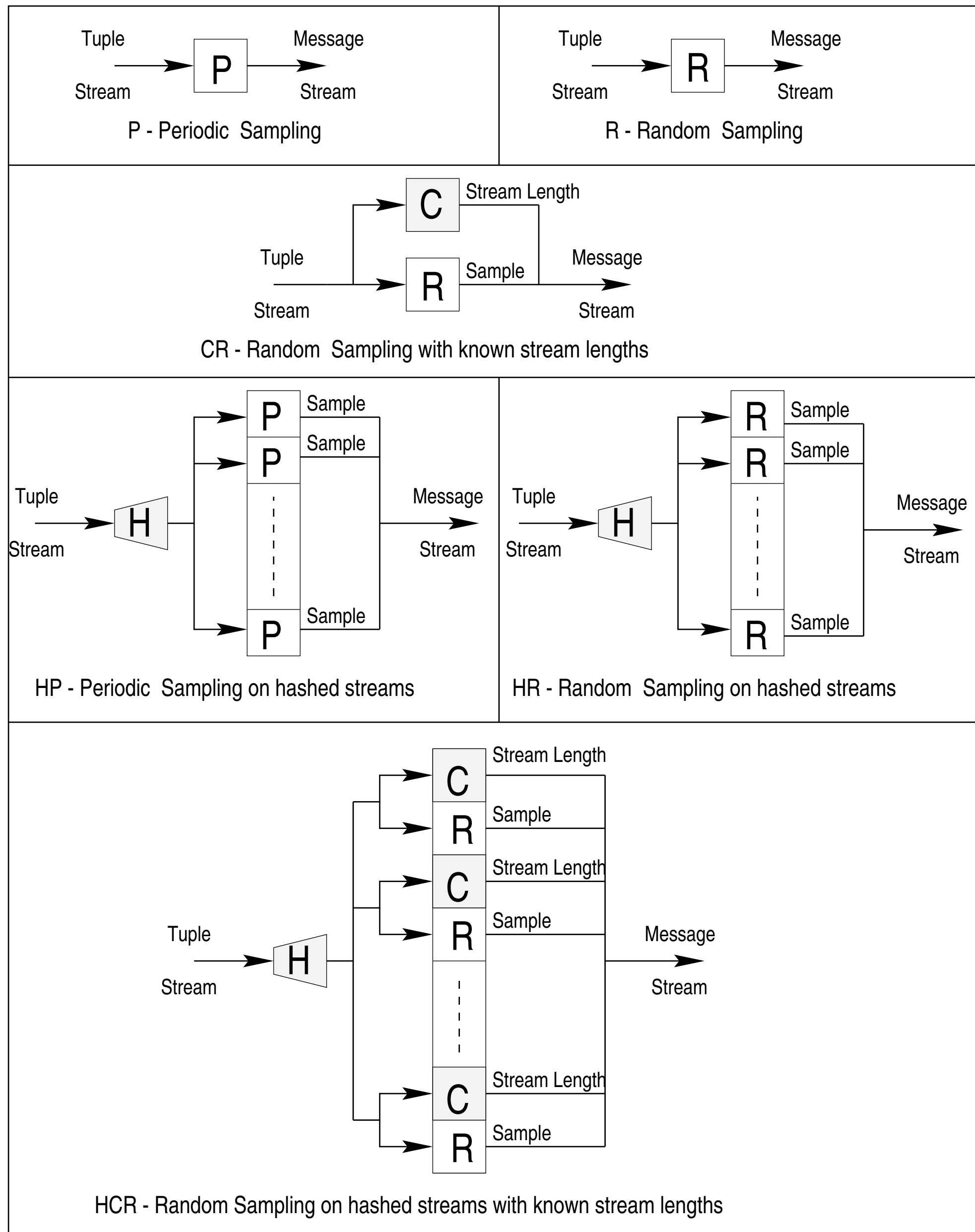

Figure 4.3. Six samplers built from the basic components.

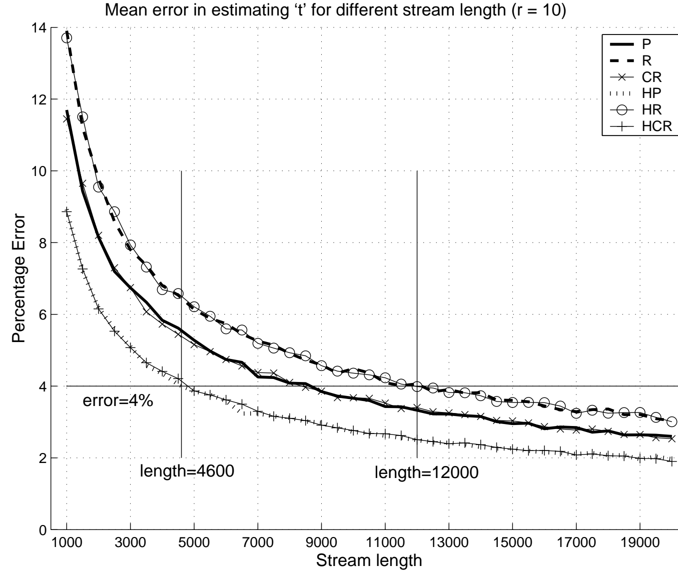

Figure 4.4. Estimating the frequency of a tuple: Mean error for different input stream lengths.

stream length, the error decreases.

Note that all the samplers in Figure 4.4 have the same sampling rate and hence incur the same overhead. It is therefore interesting to compare their convergence rate. Assuming that we fix the maximum tolerable error at 4%, Figure 4.4 shows that R_r needs to sample almost three times longer than $H[P_r]_n$ (12000 versus 4600). This experiment shows that, with a fixed profiling overhead, the stratified periodic sampler reduces the error below a maximum tolerable error faster than a random sampler. Most importantly, experiments with real benchmarks yield similar results, as we will show in Section 4.7.

Furthermore, the graph shows that, based on their accuracy, the samplers divide into three equivalence classes. In decreasing error order, the three classes are $\{R_r, H[R_r]_n\}$, $\{P_r, CR_r\}$, and $\{H[P_r]_n, H[CR_r]_n\}$. Let us now explain why some samplers have the same accuracy. The intuition behind the equivalence of P_r and CR_r was explained in Section 4.5.1. This intuition also explains the equivalence of $H[P_r]_n$ and $H[CR_r]_n$. Perhaps more surprising is the equivalence of R_r and $H[R_r]_n$. Intuitively, it may appear that $H[R_r]_n$ should perform better than R_r. The reason why

this is not the case stems from the fact that our random sampler R_r is stateless, i.e., each element in the input stream is selected *independently* from the other elements, with a probability $1/r$. This observation allows us to think of R_r, equivalently, as if each of its input elements was directed into a separate substream (of length one), which is sampled with a separate R_r sampler. This alternative view is equivalent to $H[R_r]_n$, which explains why stratification with our proposed R_r is of no benefit.[1]

Finally, let us intuitively explain why the most successful design, $H[P_r]_n$, is superior to P_r. Consider the simple example of an input stream consisting of eight 1's and eight 0's, randomly permuted. A P_8 sampler produces a sample size of 2. Using this sample, we would estimate the ratio of 1's and 0's accurately only *half the time*. Now assume that $H[P_8]_2$ splits the input stream into separate 1 and 0 substreams. $H[P_8]_2$ will also produce two messages. But, since the substreams are fully biased, the messages will *always* convey the correct ratio of the number of 1's and 0's.

While we have shown six specific compressor designs, it should be clear that within this framework, other combinations of components are possible. A complete and systematic study of other sampling schemes is outside the scope of the current work.

4.5.4. Two-Level Compressors

While sampling techniques are lossy stream compression methods, a straightforward lossless stream compression method can be built using a cache-like associative table of counters that accumulate frequency counts of the input tuples. When a table entry must be replaced, or when the counter reaches a maximum value, the tuple is dispatched to software with its corresponding frequency count. Although this method works, it is hardware-intensive. First of all, the table entries can be quite wide. Assuming a 4-byte word (32 bits), two words of profiling information per tuple

[1] Stratification of the input population improves performance when the size of the subpopulations is known ([43], CR_r, P_r) or when the samples are not picked independently (P_r, countdown random sampling used in [9, 36, 66]).

and a 1-byte counter yields 9 bytes per entry. Furthermore, to avoid frequent tuple replacements (and frequent communication with software), the table needs to capture the working set of the profiling application. For a table with as few as 4K entries the total is 36 KB of storage. With 64-bit words, this grows to 68 KB. Finally, accessing such a table associatively would require compare/match logic the width of the tuples being held in the table. Hence, any practical implementation of a compressor would not use this straightforward approach. However, we will show that the associative counter table is nevertheless a useful profiling component besides being an useful starting point for exploring other compression schemes. We use A_k to denote an associative counter table with k entries.

While the associative counter table alone is an unrealistic compressor, it can be combined with the samplers we presented in the previous section as a second-level compressor. Compressing the messages generated by the samplers will further reduce overheads without affecting accuracy. The accuracy is not affected since the compression performed by A_k is lossless. Therefore, the $H[P_r]_n A_k$ and the $H[P_r]_n$ compressors have equivalent error characteristics, but the $H[P_r]_n A_k$ compressor incurs lower overhead than the $H[P_r]_n$ compressor.

Furthermore, since the tuple stream that is input to the A_k component is already a compressed stream, the compression requirements of the A_k component are not stringent. Therefore, a table as small as 16 entries suffices to achieve a further compression ratio between 1.15 and 2.5 for our benchmarks.

4.5.5. Stratified periodic sampling

We now look at the stratified sampling scheme ($H[P_r]_n$) in greater detail.

Figure 4.5 shows the design of a stratified sampler. Each cycle, a tuple is picked from a tuple queue (that absorbs burstiness in the incoming tuple stream). The hash function computes a *signature* of the tuple. The signature is used as an index to select a counter in the counter table. The selected counter is then incremented. If the counter reaches its maximum threshold value r, the counter is

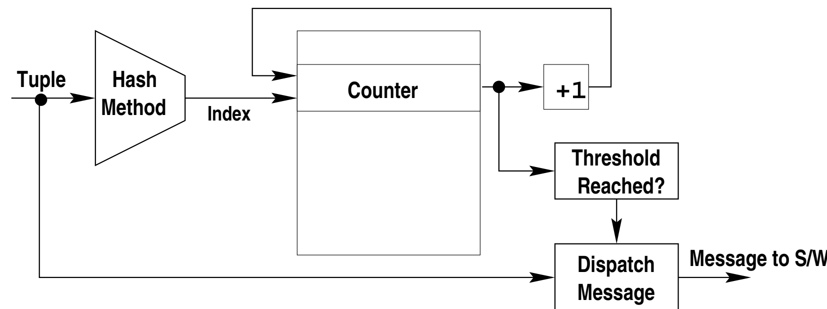

Figure 4.5. Stratified Sampling Technique

reset to zero, and a message consisting of the complete tuple (with an implied occurrence count of r) is sent to profiling software via the output queue.

In our implementation, we use the following hash function. Given a tuple `<pc, w>`, the index is computed as follows:

```
npc = flip(randomize(pc));
nw = randomize(w);
index = xor-fold(npc xor nw, index-size);
```

The function `randomize(w)` looks up a 256-entry random number table for each of the individual bytes of w and composes the new bytes together. `flip(w)` reverses the bytes of w. `xor-fold(w, n)` splits `w` into `n`-bit chunks and `xor`s the chunks together.

We used this elaborate hashing function since it provides the best accuracy of all the hashing functions that we experimented with. We have not systematically studied different hashing functions and the trade-offs between profile accuracy and hardware cost. In practice, an actual implementation of the stratified sampler might use a cheaper hash function.

4.5.6. Hardware cost of the stratified sampler

The biggest cost of the stratified sampler is the counter table. Assuming a 2K-entry counter table, and a 8-bit counter (which can support a sampling rate between 1 and 256), the country table requires

2KB of hardware space. Assuming an implementation of a hashing function described above, we require an additional 256 bytes for encoding the random number table. Besides this, additional space is required for the input and output buffers. Assuming an 128-entry input buffer with 2-word tuple entries, we require another 1KB. Assuming that we buffer 128 messages before raising an interrupt, we need another 1KB space for the output buffer. Thus, we require a total of 4.25KB for this design. This is a small hardware cost because the stratified sampler can be placed away from the processor core.

4.6. Reducing Collisions: Adding tags to stratified sampling

The stratified periodic sampling described in Section 4.5.3 makes no effort to resolve tuples that hash to the same substream. At the other extreme, the associative table of counters described in Section 4.5.4 avoids *all* aliasing, by comparing tuples against complete tags. Unfortunately, the latter design is relatively expensive. This section presents a compromise solution that reduces (but does not eliminate) aliasing by maintaining *partial tags*.

4.6.1. Design detail

In this design, the signature generated by the hash function has more bits than are required for indexing into the table. The more additional bits, the better the ability to discriminate amongst tuples. The signature is subsequently divided into an index and a tag.

Given a tuple `<pc, w>`, the index is computed as described in Section 4.5.5. The tag is computed as `tag = xor-fold(pc xor w, tag-size)`. The index is used to select a table entry; if the tags match, there is a hit in the table, otherwise there is a miss. Like a cache memory, the table can be direct-mapped, set-associative, or fully-associative.

Each entry in the table contains the tag, a hit counter, and a miss counter. The hit counter keeps track of the number of occurrences of a tuple. The miss counter is used in making replacement

decisions and is discussed below, following an informal description of the replacement process.

At some point, it becomes necessary to evict a tuple's entry from the table, either because its hit count has reached a maximum threshold or because another tuple (with a different tag) maps to the same table entry. If it reaches the maximum threshold, the current tuple is reported to software. In the case of an eviction, the replaced tuple and its occurrence count *should* be passed to software, but this is not possible because only the hashed signature is available in the entry, not the complete tuple. This problem is solved by deferring eviction and placing the to-be-evicted entry into an *eviction state* in which it waits for the same signature to be seen once again.

If the to-be-evicted tuple occurs frequently, then it is likely to occur again soon, and the entire tuple is available so complete information can be passed to the software. If the to-be-evicted tuple occurs only rarely, and it does not occur again soon, its value will eventually be discarded.

Figure 4.6 shows a state machine diagram that describes the detailed operation of a table entry, including the states that an entry goes through, and conditions under which an entry is allocated, evicted, or replaced.

Initially, all entries in the table are `Empty`. When an incoming profile tuple accesses an entry, the entry is allocated for the tuple and the entry transitions to a `Valid` state (Transition E1). In this state, the entry accumulates matching tuples in the hit counter (Transition E3). When the hit counter saturates, a summary message containing the counter value and the tuple is sent to the message queue and the entry transitions to `Empty` (Transition E2).

If there is a miss in the `Valid` state, some other tuple is seeking access to the entry. Because the necessary tuple information for the to-be-evicted entry is not available, the entry transitions to an `Evict` state (Transition E4). The entry is left in the `Evict` state until the matching tuple is seen again (Transition E6). However, the state machine waits until "enough" hits have accumulated in the counter (Transition E6a). This reduces the flood of messages that can be caused by repeatedly aliasing tuples. The *ShouldEvict* condition in the state diagram specifies the exact value of "enough".

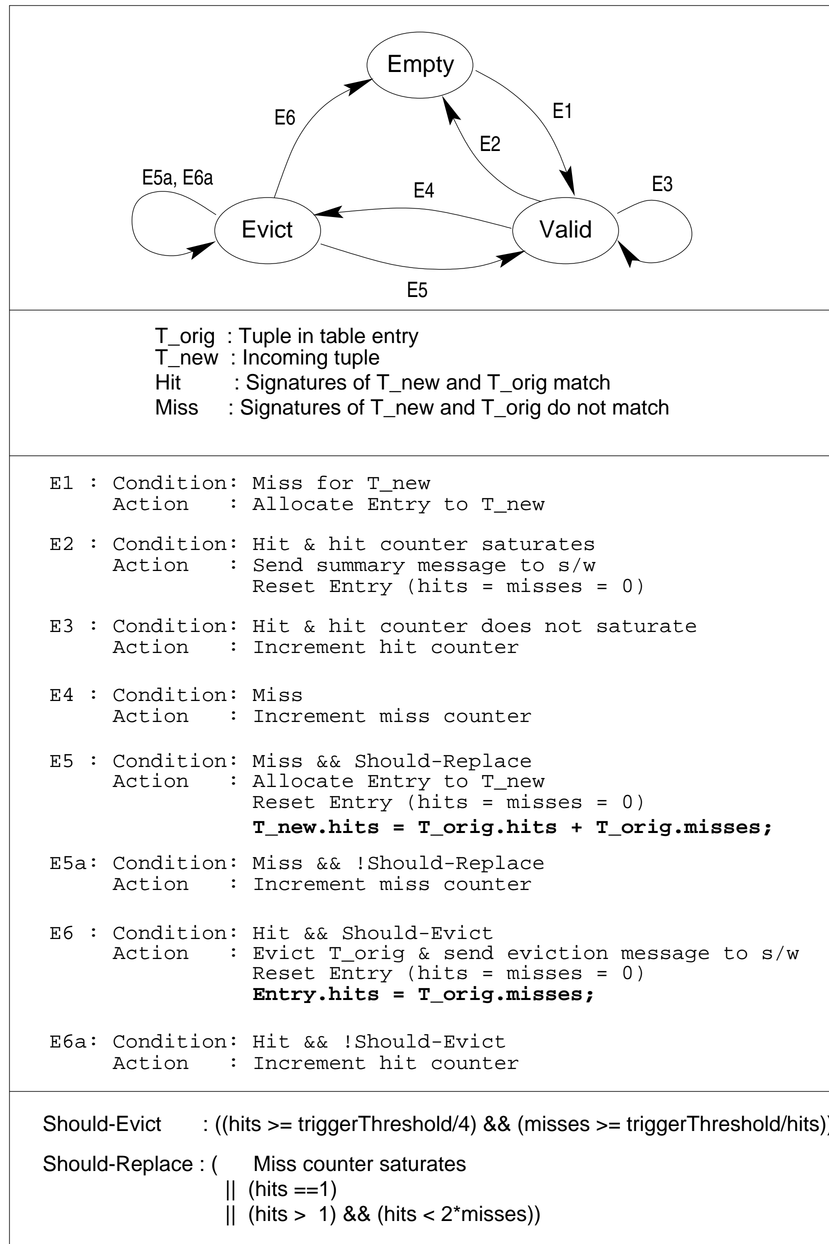

Figure 4.6. State diagram describing the states of a profile entry

In the `Evict` state, misses to the entry will accumulate in the miss counter (Transition E5a), but if “too many” misses are accumulated, the state machine gives up and replaces the current tuple with the new conflicting tuple (Transition E5). In this case the original entry is lost. The *ShouldReplace* condition in the state diagram specifies the exact value of “too many”. This replacement policy is similar to the replacement policy in the value profiling algorithm presented in [34].

The unusual manipulation of the hit and miss counters by Transitions E5 and E6 illustrates an important point – it is important not to drop counts when assigning the entry to a new tuple. We found after extensive experimentation that resetting counters almost always leads to lower accuracies because valuable information concerning the input stream is discarded. For every entry, the hit and miss counters together maintain the length of the stream that access that entry. The examples in Section 4.5.1 implied that it is this exact knowledge of the stream length that leads to increased accuracy. Resetting the hit/miss counters to zero would obscure this valuable information.

Transition E5 represents replacement of a previous tuple by a new tuple. The hit counter is *not* reset as might be expected. Instead, the hit count of the evicted tuple is assigned to the new tuple along with the miss count. Although the hit count does not belong to the new tuple, this avoids losing track of the stream size.

During Transition E6, we initialize the hit counter of the entry with the miss count of the evicted entry. The next tuple to hit in this entry (hopefully the tuple that caused the miss count to be incremented via Transitions E4 and E5a) will assume the new count. Again, the exact stream size is maintained.

4.6.2. Hardware cost

The tagged profiler incurs extra space overhead beyond the stratified sampler. Besides the hit counter, every entry requires a tag, a miss counter, and bits to code the current state of the FSM. Using a 1-bit tag to encode the FSM state (the Evict state can be inferred by a non-zero value in the

miss counter), a 3-bit miss counter, and a 1-bit tag, this is 5 extra bits per counter. With the same sized counter table, and a 8-bit hit counter, this is at least a 62% increase in space requirements over the stratified sampler, in addition to the logic required to implement the state machine.

4.7. Experimental Results

We evaluate the proposed profiling hardware designs through a timing simulation of the value profiling application described below. The timing simulation is needed for measuring the profiling overhead.

4.7.1. Example Application: Load Value Profiling

A number of micro-architectural studies have demonstrated that programs exhibit significant *value locality*, the phenomenon that a small number of values occur repeatedly in the same register or memory location [45, 55, 64, 79, 84]. In Chapter 2, we discussed how this phenomenon leads to computational reuse opportunities. While there exist different techniques for exploiting computation reuse, all techniques rely on some kind of value profiling to identify value locality of instructions [22, 29, 71, 73]. This dissertation also relies on collecting two forms of value profiles. In this dissertation, we focus on load value profiling to evaluate the stratified sampling profiler and compare it with other profilers. A load value profile computes the frequency distribution of values loaded by different load instructions.

4.7.2. Methodology

We used the Simplescalar toolset [37] to model a 4-way machine with 64-KB L1 data and instruction caches, 1-MB unified L2 data cache, and a gshare branch predictor. The cycle-level timing model is used for computing profiling overheads and does not affect the actual sampling algorithms.

We use a collection of SpecINT95 benchmarks and Java programs listed in Table 4.7.2. The

SpecInt95 benchmarks were compiled for the SimpleScalar ISA by *gcc* with optimization flags "-O3". The Java programs (including *strata*) were compiled by *strata* [83], a Java-bytecode to native-ISA compiler for the SimpleScalar ISA. For all programs, simulation was performed after skipping the initialization phases. For the SpecInt95 benchmarks, the recommendations of Sherwood and Calder [82] were used in determining the simulation starting points. For the Java benchmarks, the starting points were determined empirically (by examining the source code, knowledge of benchmark and output, and experimentation).

Benchmark	Comment	Input
go	SpecInt95	5stone21 files (Ref)
li	SpecInt95	8-queens.lsp (Test)
m88ksim	SpecInt95	Ref input
gcc	SpecInt95	cccp.i
perl	SpecInt95	primes.pl, primes.in
raytrace	SpecJVM98	Speed 100
strata	Java-bytecode to Simplescalar-ISA compiler	Some class file
jess	SpecJVM98	speed 100
jack	SpecJVM98	Jack.jack
db	SpecJVM98	speed 100

Table 4.1. Benchmarks used to evaluate the profiling schemes

4.7.3. Evaluation Metrics

Profiling Error

The errors in our value profiles is computed using an ideal value profile. Examining the same stream as our profilers, the ideal profile accumulates *all* generated events, rather than just the samples.

When computing the error, we remove from both profiles all (*load, value*) tuples that are highly unlikely to be used by a realistic value-reuse optimizer. Taking these tuples into account would introduce error that is irrelevant to the optimizer. Analogous to the error metric in [34], from both profiles, we discard loads that execute infrequently, as well as values that are infrequent for a given

static load. Namely, we select for profiling a static load only if it executes at least 1000 times. Furthermore, only sufficiently invariant tuples are selected. A (*load, value*) tuple is *sufficiently invariant* if it accounts for at least 10% of the executions of the load. Finally, we select only those loads that have at least 40% of their dynamic execution accounted for by sufficiently invariant tuples.

The profiling error is computed as follows. Let $v = (pc, val)$ be a tuple that has been selected from the ideal value profile. The ideal invariance of v is computed as $I_i(v) = n_i(v)/n_i(pc)$ where $n_i(v)$ is the number of times v occurs in the ideal profile, and $n_i(pc)$ is the execution count of pc. Let $I_p(v) = n_p(v)/n_p(pc)$ be the tuple's invariance estimated by our profiler; $n_p(v)$ and $n_p(pc)$ denote the number of times our profiler sees the tuple v and the load pc, respectively. Then, the error in profiling the invariance is $e(v) = |i_i(v) - i_p(v)|$. We compute the error for the entire profile by taking a frequency-weighted average of $e(v)$ over all selected tuples, i.e., the error $e = \sum f(v)/f \times e(v)$ over all selected tuples v, where $f(v)$ is the cumulative execution frequency of v, and f is the execution frequency of all selected tuples.

Profiling Overhead

Our evaluation assumes interrupt-driven communication between the compressor and the software profiler (see Section 4.4.2). We use an analytical model to compute profiling overhead. The overhead is dependent on:

- *Per-interrupt fixed costs*: This is a fixed cost that depends on the OS and the specific processor. Anderson *et al* [9] show that for their system, per-interrupt fixed costs are about 214 cycles. We use this value in our model.
- *Number of messages processed per interrupt*: In our model, we assume at least 100 messages per interrupt to amortize the per-interrupt fixed cost. This requires an output buffer that can buffer at least 100 messages.

- *Processing time per message*: This is the time required to fold the message into the profile; the actual value is specific to the profiling application. We assume a fixed cost in our model, as described below.

In their paper, Zilles and Sohi [34] state that with careful assembly coding of their interrupt handler, every message can be processed in 10-30 cycles. Because we do not perform convergent profiling checks as in [34], at least 3 cycles per message are saved. On the other hand, by processing at least 100 messages per interrupt, we incur under 3 cycles in fixed interrupt costs per message (214 cycles per 100 messages). Therefore, in the worst case, we assume the interrupt overheads will be 30 cycles per message. Based on this number, we estimate the profiling overhead to be $Overhead = 30 \times NumMessages$, where $NumMessages$ is the number of messages dispatched. The percentage overhead is computed with respect to the total simulation time.

4.7.4. Evaluating compressors

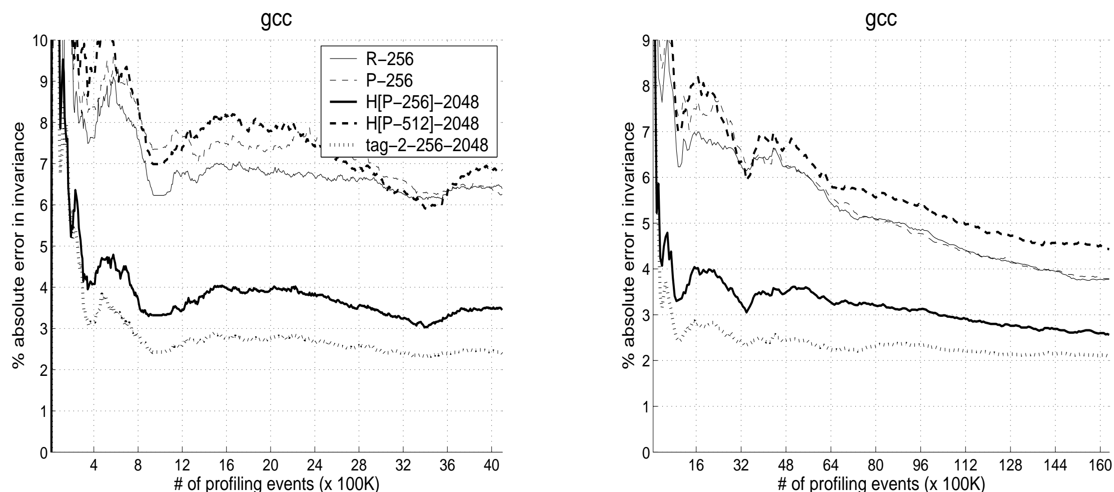

Figure 4.7. Results for *gcc*: The first graph shows the variation of % error with program progress (up to 4M events). The second graph shows errors for a longer duration (up to 16M events).

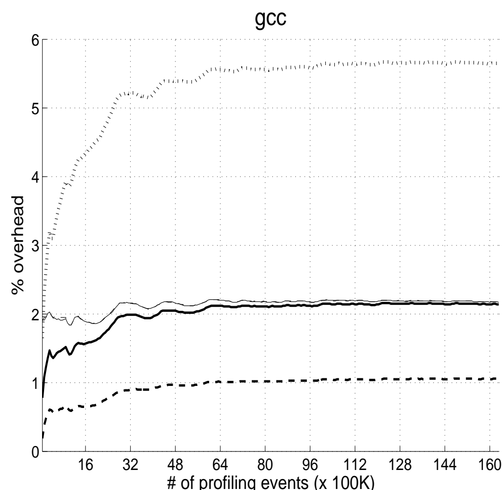

Figure 4.8. Results for *gcc*: The graph shows the variation of cumulative % overhead with increasing time (up to 16M events). The plots for the R_{256} and P_{256} compressors overlap.

In this section, we evaluate the following compressors: R_{256}, P_{256}, $H[P_{256}]_{2048}$, $H[P_{512}]_{2048}$ (presented in Section 4.5.3), and a 2-bit tagged compressor (presented in Section 4.6) with an 8-bit hit counter (equivalent to a sampling rate of 256) and a 2-way set associative 2048-entry counter table. We do not present results for other tagged compressors since our results indicate that for our tagged implementation, the 2-bit tagged compressor has the best error-overhead behavior among all tagged compressors.

Figure 4.7 shows for *gcc* the variation of % error with program progress. The first graph is plotted for 4M profiling events (27M instructions). The second graph is plotted for 16M profiling events (105M instructions). Figure 4.8 shows the variation of % overheads with program progress. We selected *gcc* because it is one of the hardest programs to profile and has a much bigger working set than the other programs.

We first compare the random and periodic samplers. The graphs show that the two are almost identical in performance both in error and overhead. It is interesting to compare these results with the

Monte Carlo simulation results presented in Section 4.5.3. For randomly generated input streams, Monte Carlo simulations showed that the periodic sampler performs better than the random sampler. However, in practice, programs do not generate random input streams and this seems to explain why P_{256} does not perform better than R_{256} on real workloads. For the rest of the discussion, we discuss the random profiler only.

Next, we discuss the tagged compressor. The error graphs show that the tagged compressor has the best accuracy of all compressors, but the low error comes at the cost of almost three times the profiling overhead, as shown in Figure 4.8. Furthermore, from the graphs, we conjecture that for *gcc*, the improvement in accuracy over the much simpler stratified sampler is not significant enough to merit the higher overheads and the higher hardware complexity of the tagged compressor design. At this juncture, we wish to point out that the tagged compressor design presented in Figure 4.6 is only one of many possible implementations of a tagged compressor. There could be other implementations (not examined in this dissertation) that could perform better.

We now compare the $H[P_{256}]_{2048}$ and the R_{256} compressor. The graphs in Figure 4.7 show that the stratified sampler outperforms the random sampler in its accuracy. Assuming an arbitrary maximum tolerable error threshold of 4%, the graphs show that the $H[P_{256}]_{2048}$ profiler reaches this error threshold after about 200K events whereas R_{256} reaches this threshold after about 1.4M events – seven times longer than the stratified sampler.

The overhead graph shows that the stratified sampler always has a lower overhead than the random sampler because the $H[P_{256}]_{2048}$ compressor retains up to $256 \times 2048 = 512K$ tuples in the counter table, which means that up to $2K$ fewer messages are dispatched to the software when compared to the random sampler. However, if the programs are profiled for a long time, this advantage vanishes as shown by the converging overhead plot.

Let us now examine how to reduce profiling overheads for the stratified samplers while maintaining the same accuracy as a random profiler. If we compare the $H[P_{512}]_{2048}$ and the R_{256} com-

pressors, we find both have similar accuracy, but the $H[P_{512}]_{2048}$ incurs half the overhead when compared to R_{256}. This shows that with the stratified sampler, we can achieve the same accuracy as a random sampler at a lower sampling rate, and hence at lower profiling overheads. The factor by which the sampling rate can be reduced is benchmark-specific as can be seen from Figure 4.9. The graphs show that, except for *go*, the sampling rate can be reduced by at least a factor of two while maintaining the accuracy level of a random sampler.

One final conclusion that we can draw from the error graphs is that the stratified sampler and the tagged compressor stabilize more quickly than the random and periodic samplers, i.e. they converge to their “final” errors much more quickly than the random and periodic samplers.

By using profile convergence checks and instruction filtering techniques proposed in [34], the performance of the stratified sampler can be improved further. Instruction filtering reduces aliasing in the hashed substreams and can lead to faster convergence of profiles.

The preceding discussion focused on *gcc*. Figure 4.9 shows the error curves for the five compressors for all the other benchmarks. A closer examination of the graphs for *db* and *perl* show that both these graphs do not have an error plot for the periodic sampler (P_{256}). This is because the errors are significantly higher than those for all the other samplers.

Figure 4.10 shows the plots for *perl* and *db* – these are similar to that shown in Figure 4.9 except for the range of the Y-axis. The larger range shows that the errors for the periodic sampler is indeed quite high when compared to the other profilers. These higher errors for *perl* and *db* could be because the sampling period aligns with some periodic behavior of the program which leads to large inaccuracies.

4.7.5. Sensitivity study of the stratified sampler

We now present results of a sensitivity study of the stratified sampler by considering four different table sizes (512, 1024, 2048, and 4096). Figure 4.11 shows error plots for the $H[P_{256}]_{512}$,

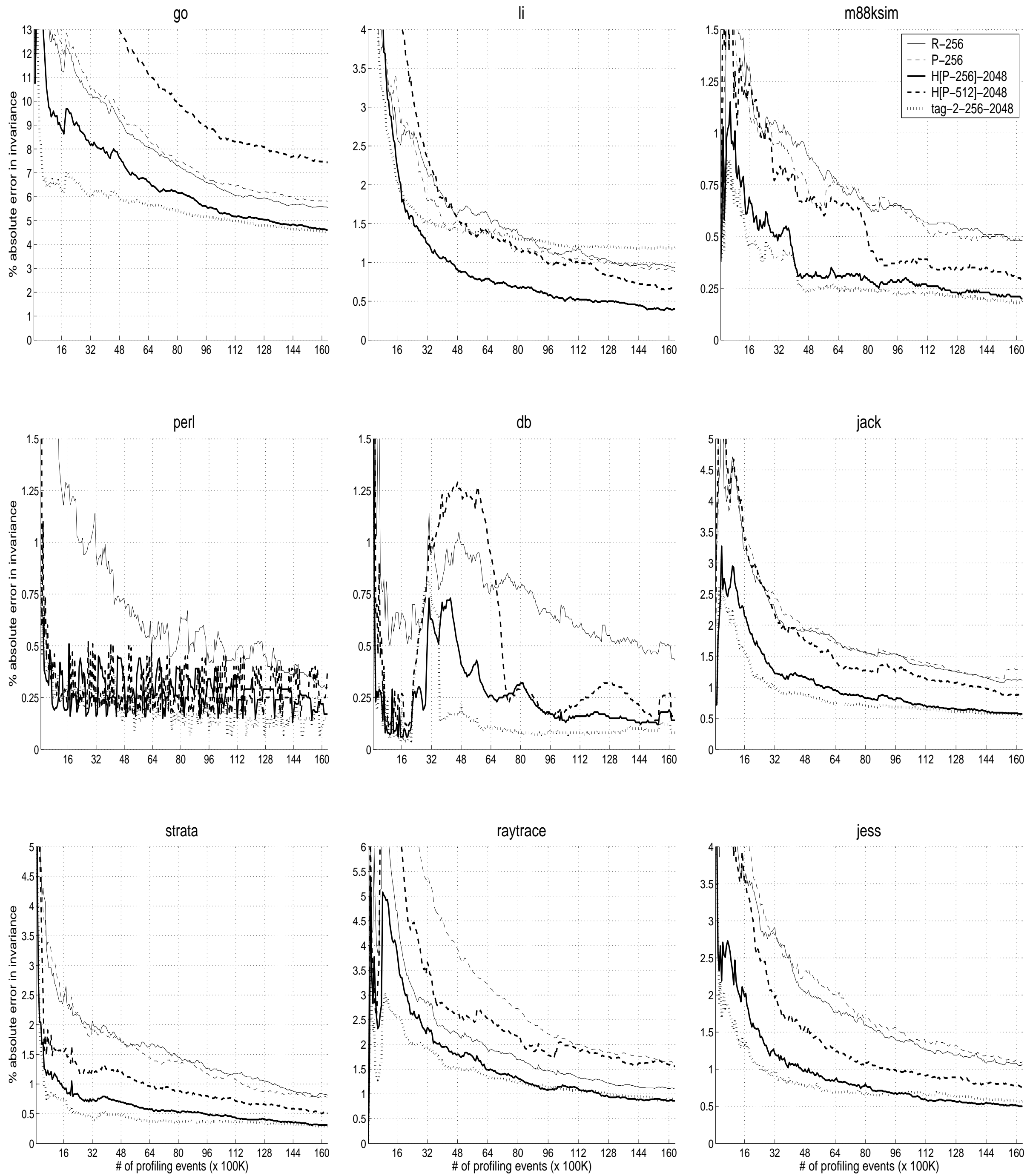

Figure 4.9. Results for all the other benchmarks: The graphs show the variation of % error with program progress (up to 16M events).

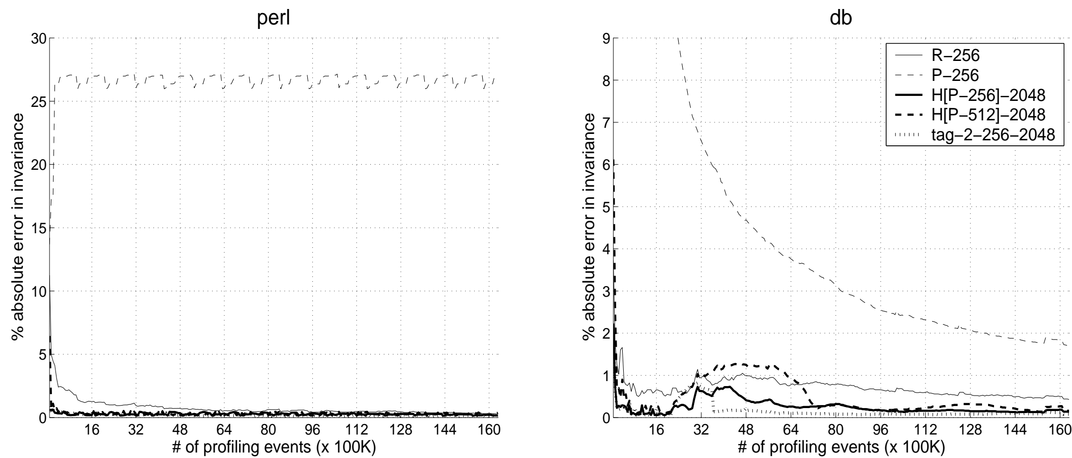

Figure 4.10. Large profiling errors for P_{256} due to periodicity in program behavior – for *db* and *perl*

$H[P_{256}]_{1024}$, $H[P_{256}]_{2048}$, and $H[P_{256}]_{4096}$ compressors for *gcc* and *raytrace*.

The graphs show that with increasing number of table entries, the performance gets better but only *after* a sufficient number of tuples have been seen. Because the counter table accumulates tuples that are not sampled until the counters overflow, during the initial phases, when the ratio of stream length to the output messages is small, more messages are sent out from a smaller table. Hence, for some benchmarks (*raytrace*, for example, shown in Figure 4.11) the accuracy is better with fewer counters during the initial phases. As the program progresses, this effect diminishes.

4.7.6. Two-level compression

If a small associative counter table is added, additional reductions in overhead can be achieved. Figure 4.12 shows the profiling overheads for the $H[P_{256}]_{2048}$ profiler with and without a 16-entry associative buffer. First, the graph shows that except for `db`, the $H[P_{256}]_{2048}$ stratified sampler incurs under 6% overhead. Second, the graphs shows that by adding the second compressor, there is a reduction of overheads across the board for all benchmarks. The augmented stratified sampler incurs less than 4.5% overhead for all benchmarks.

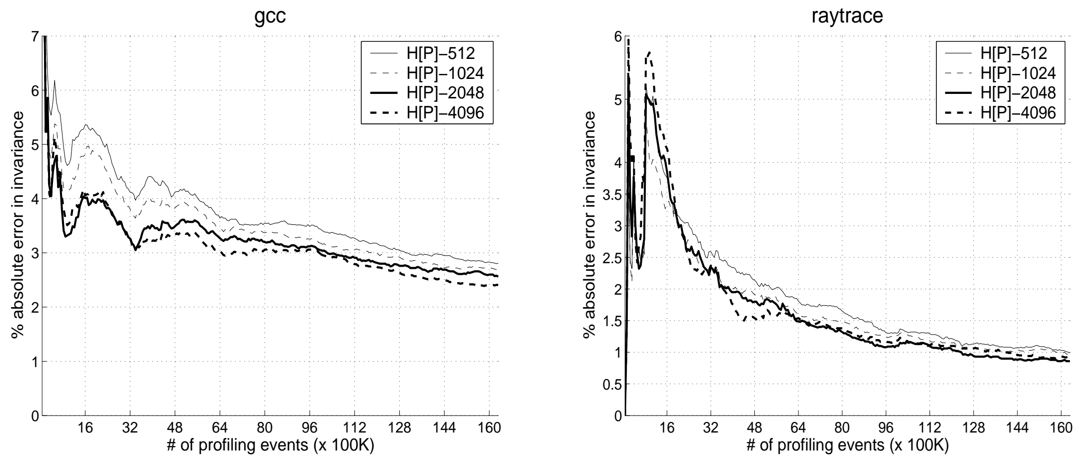

Figure 4.11. Sensitivity results of the stratified sampler for four different table sizes for *gcc* and *raytrace* (up to 16M profiling events)

The overhead reduction due to the counter table is pronounced for *m88ksim, jack* and *strata*. For these programs, the output message stream is dominated by a few prominent tuples. It is these repetitive output messages that enables the high second-level compression of the message stream. This result shows that the associative array of counters is a useful component in building efficient compressors.

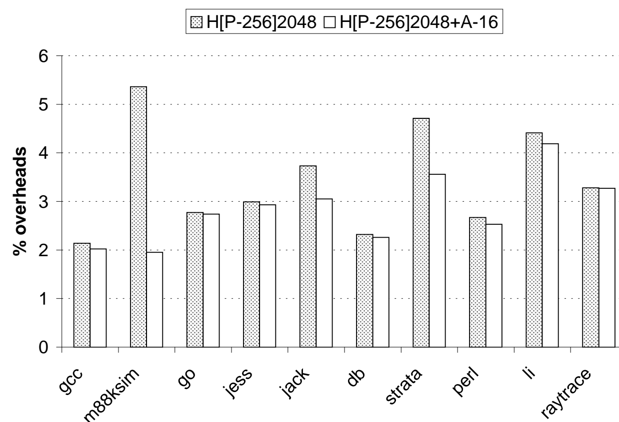

Figure 4.12. Reduction in overheads due to the A_{16} component

4.7.7. Simultaneous profiling: Edge and Call target profiling

In Section 4.4.1, we stated that the hybrid profiler can collect multiple profiles simultaneously. To demonstrate the feasibility of this application, we collected call-target and edge profiles simultaneously using the $H[P_{256}]_{2048}$ compressor. These profiles are used by the runtime optimizer to select inline candidates and to build traces, respectively. Call target profiles are especially important for Java programs because, unlike C programs, the presence of virtual calls in Java makes it difficult to statically determine the call targets at a call site. Call target profiles enable the optimizer to determine the likely call targets and inline sites and implement optimizations such as feedback-directed inlining [11] and polymorphic inline caching [54]. Edge profiles enable the optimizer to build traces

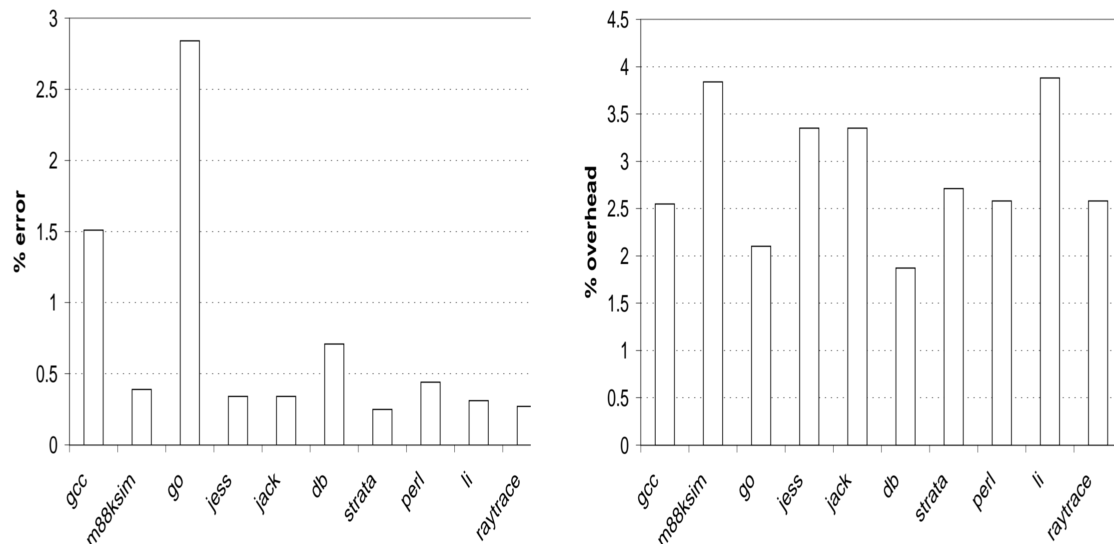

Figure 4.13. Final error and cumulative overhead for simultaneous call and edge profiling after 4M profiling events

Figure 4.13 shows the results of simultaneously collecting edge and call target profiles. The graphs show that for all benchmarks, the error is less than 3%. For six of the benchmarks, the error is negligible. Furthermore, the overheads are also very low; the profiling overheads are under 4%

for all benchmarks. The accuracy is significantly better than the accuracy in the preceding value profiling application because branches, jumps, and calls take values from a much smaller value set than load instructions.

4.8. Summary

Profile-driven optimizations require flexible profiling support that can collect a variety of profiles accurately, rapidly, and with low overheads. In this chapter, we presented a hybrid hardware-software profiling model in which the hardware compresses the input profile stream to generate a smaller stream of output messages that is processed by software to construct the required profile. Since optimizations can tolerate profiling errors, we show that the compression can be lossy in that the occurrence counts reported for tuples in the input stream need not be exact.

On the basis of the lossy compression metaphor, we presented a framework of profiling components that can be composed in multiple ways to build hardware compressors. Conventional random sampling, periodic sampling, and stratified sampling were proposed and studied. The stratified sampling scheme uses a hashing scheme to split the input stream into multiple disjoint streams that index into a table of counters. In addition, we proposed a more sophisticated compression scheme that builds on the stratified sampling scheme by using tags to detect aliasing in the counter table.

We used load value profiling as an example application for evaluating the proposed compressors. We showed that the stratified sampling scheme has the best error-overhead performance among all the compressors we studied. For *gcc*, we showed that with the same or smaller profiling overheads, the stratified sampler achieves a desired accuracy twice as fast as a random sampler. We also showed that for *gcc*, if the profiling time and the desired accuracy level are fixed, the stratified sampler can achieve these thresholds with half the overheads as a random sampler. The overhead factors are benchmark-specific (more than two for *perl*, *db* and other benchmarks).

We also showed that while the tagged compressor has better accuracy than the stratified sampling

scheme, the improved accuracy comes at significantly higher overheads (twice or more) and higher hardware complexity.

We then proposed an enhancement to the different compression schemes by introducing a second-level lossless compression that can further reduce profiling overheads without affecting accuracy. We show that additional lossless compression between 1.15 and 2.5 can be achieved by using an associative buffer of 16 entries that summarize the messages from the first level compressor.

CHAPTER 5

IDENTIFYING SPECIALIZATION SCOPES

Section 1.3 presented the key challenges in implementing a specializer transparently within a dynamic optimizer. One of the key challenges is determining the code regions (called specialization scopes) that can be profitably specialized. This chapter presents an algorithm that identifies profitable specialization scopes.

5.1. Outline of algorithm `BuildScopes`

In Section 3.7, we presented the central idea of the algorithm using an example. For each program point `p` that accesses an object `o`, the algorithm computes a scope starting at `p` and having `o` as the key. To do this, the algorithm essentially tries to construct a skeleton of instructions (rooted at `p` and connected by data-flow dependencies) that access invariant memory locations and identifies a single-entry, multiple-exit control-flow boundary that encloses this skeleton.

In this section, we present an outline of the algorithm and fill in the details in the rest of the chapter.

Input: Given a candidate method `m`, the algorithm requires the CFG as well as the SSA form of `m`. The loops of the CFG are expected to be in canonical form where loop headers have exactly one entry edge and exactly one back edge. The SSA dataflow graph is also expected to be in canonical form where all constant assignments and copies have been propagated and eliminated.[1]

Output: Given a candidate method `m`, the algorithm identifies a set of specialization scopes `S(m)` within the method for which specialized code is generated. These scopes satisfy the following properties.

- Every scope in `S(m)` is a single-entry, multiple-exit region of the CFG.

[1] In practice, the algorithm handles copies.

- No two scopes in `S(m)` overlap with each other. As a special case, no two scopes in `S(m)` can be nested within each other. In principle, while nested scopes can be implemented, this dissertation does not consider nested scopes for the sake of simplicity of the scope-building algorithm.

Algorithm Outline: Conceptually, the algorithm consists of the three steps shown in Figure 5.1. In

```
Scope Set BuildScopes(Method m)
{
        /* 1. Build candidate scope skeletons */
   skeletons = BuildScopeSkeletons(m);  /* Figure 5.9 */

        /* 2. Identify single-entry, multiple-exit scope boundaries */
   scopes = {};
   for each (Skeleton sk ∈ skeletons){
      Scope s = IdentifyScope(sk);  /* Figure 5.16 */
      Add s to scopes;
   }

        /* 3. Identify non-overlapping/non-nested scopes */
   return ResolveConflicts(scopes);/* Section 5.4 */
}
```

Figure 5.1. High-level outline of the `BuildScopes` algorithm

the first step, the algorithm identifies a set of scope skeletons. In the second step, these skeletons are converted into single-entry, multiple-exit specialization scopes. In the final step, the algorithm uses a cost-benefit model to identify a subset of these scopes that are non-overlapping, non-nesting, and profitable.

Skeletons of a scope are made up of instructions which can be eliminated using the knowledge of the value of the lookup key of the scope, i.e. if an instruction `I` belongs to a scope `S(k)`, then, the output of `I` is computable from the value of `k`. However, a scope contains instructions that might not be specializeable (in addition to all the instructions in the skeleton). It is a CFG region that satisfies the single-entry, multiple-exit property.

5.1.1. Defining an acyclic scope skeleton

We first define a scope skeleton on acyclic SSA dataflow graphs and present an algorithm to build acyclic scope skeletons. We use this simplified problem context to highlight the design decisions made in developing the scope-building algorithm. These same design decisions are part of the complete scope-building algorithm that also deals with cyclic SSA graphs.

A scope skeleton of an SSA node p is essentially a forward slice of the SSA graph seeded at p and extending only upto those SSA nodes that are specializeable, i.e. those that access invariant locations of objects and SSA nodes that represent invariant scalar computations. A specialization scope is essentially just a single-entry, multiple-exit CFG region that encloses the skeleton. Thus, a scope skeleton is defined on SSA graphs, whereas a scope is defined on CFGs. Thus, the shape of a scope is determined by the skeleton – scopes can be computed only after skeletons are built.

We formalize the definition of a scope skeleton below. In order to do so, we need the following two definitions.

Definition 1: A SSA node is defined to be an *object reference node* if it represents a `getfield`, `getarray`, or `arraylength` instruction. All other SSA nodes are defined to be *scalar computation nodes*. If p is an object reference node, we use the predicate `oref(`p`)` to refer to the SSA variable that holds the object referenced by the node.

Definition 2: Let p be an object reference node. Let o = `oref(`p`)`. Let o_{hot} be the hottest object referenced by p. The value of o_{hot} is obtained by querying the object-access profile. p is considered an *invariant object reference node* if p accesses an invariant memory location of o_{hot} (object field or array element). The invariance of the accessed memory location is verified by querying the store profile.

In the rest of this chapter, we will use the following three terms interchangeably: SSA node, instruction, and program point since every SSA node encodes a single program instruction, and every program instruction is associated with a program point.

Given these two definitions, a scope skeleton in an acyclic SSA graph is defined as follows. Let p be an invariant object reference node. Then, $S_{sk}(p)$, a scope skeleton seeded at p (called the *skeleton root*) is a forward slice of the SSA graph. If $S_{sk}(p)$ contains a node x, then x satisfies one of the following conditions:

- $x == p$.
- x is an invariant object reference node and all its input nodes are in $S_{sk}(p)$.
- x is a ϕ-node, all its input nodes are in $S_{sk}(p)$ and compute the same value along all incoming edges.
- x is a non-ϕ scalar computation node, and all its input nodes are in $S_{sk}(p)$.

Note that if x belongs to $S_{sk}(p)$, then, the output of x is computable at specialization time using the value of `oref(`p`)`.[2]

The above definition is a constructive definition and it is straightforward to design an algorithm to build a scope skeleton starting at an invariant object reference node. This algorithm is presented in Figure 5.2.

5.1.2. Building acyclic scope skeletons

For every invariant object reference node, `BuildAcyclicScopeSkeletons` builds a maximal scope skeleton. A scope skeleton is maximal if it cannot be grown any further. The algorithm uses the membership conditions presented earlier to add new instructions to a scope skeleton. In order to add a `getarray` instruction to the skeleton, the algorithm has to determine the array element being accessed and verify its invariance. As shown in Figure 5.3, the algorithm uses `vMap` (a mapping from SSA variables to values) for this purpose. Whenever an instruction is added to the skeleton, its output value is added to `vMap`.

---

[2]This is not always true as we will discuss later on in this section.

```
ScopeSkeletons BuildAcyclicScopeSkeletons(Method m)
{
    Let SS = {};  /* Set of scope skeletons */
    Let N  =  Set of all SSA nodes in m's SSA graph;
    while (N ≠ {}) {
        Remove a node n from N;
        Let S = BuildScopeSkeleton(n); /* Line A */
        if (S ≠ null) add S to SS;
    }
    return SS;
}

ScopeSkeleton BuildAcyclicScopeSkeleton(SSA-Node n)
{
    if (n is not an object access node) return null;

    Let vMap = {}; /* Mapping from SSA registers to values */
    if (!IsInvariantObjectAccessNode(n, vMap)) return null;

    Let N = {};       /* Work list of SSA nodes */
    Let S = {n};      /* instructions in the skeleton */
    Add all target nodes of n to N;
    while (N ≠ {}) {
        Remove a node x from N;
        if (all input nodes of x are in S) {
            if (     (x is an object access node)
                 && (IsInvariantObjectAccessNode(x, vMap)))
            {
                Add x to S and add all target nodes of x to N;
            } else if ((x is a φ-node) and (all inputs of x compute the same value)) {
                vMap(outputRegister(x)) = ComputeOutput(x, vMap);
                Add x to S and add all target nodes of x to N;
            } else if (x is a non-φ scalar computation node) {
                vMap(outputRegister(x)) = ComputeOutput(x, vMap);
                Add x to S and add all target nodes of x to N;
            }
        }
    }
    return S;
}
```

Figure 5.2. BuildAcyclicScopeSkeletons: Algorithm to build scope skeletons for acyclic SSA dataflow graphs

```
bool IsInvariantObjectAccessNode(SSA-Node n, Map vMap)
{
    Let oap be the object-access profile;
    Let sp be the store profile;
    Let o_hot = oap.GetHottestObject(n);
    if (n is r = getfield(o, f)) {
        if (sp.IsInvariant(o_hot.f)) {
            vMap(r) = o_hot.f;
            return true;
        }
        else return false;
    }

    if (n is r = arraylength(o)) {
        vMap(r) = o_hot.arraylength;
        return true;
    }

    if (n is r = getarray(o, i)) {
        if (vMap(i) = null) return false;
        i_val = (i is a constant) ? i : vMap(i);
        if (sp.IsInvariant(o_hot[i_val])) {
            vMap(r) = o_hot[i_val];
            return true;
        }
        else return false;
    }
}
```

Figure 5.3. Routine `IsInvariantObjectAccessNode`

While this algorithm only handles acyclic SSA dataflow graphs, the design designs behind this algorithm are also central to the scope-building algorithm that is presented later on in this chapter. These design decisions are presented below.

- *Greedy strategy:* In building maximal scope skeletons, the algorithm greedily tries to build the largest possible specialization scopes starting at the skeleton root. While a non-greedy algorithm can potentially identify better and more profitable scopes, a greedy strategy has been chosen for the sake of simplicity. Chapter 9 shows that this strategy is effective in identifying

profitable specialization scopes with low runtime overheads.

- *Hot-object approximation heuristic:* In deciding whether a node p is an invariant object reference node (Figure 5.3), the algorithm uses the hottest object referenced by p as a *representative* object referenced by p. If the accessed field in the hot object is invariant (or variant), the algorithm assumes that this property holds true for all objects accessed by p. This is a necessary approximation since it is not feasible to construct scopes (skeletons) for every possible object referenced by p. Due to this approximation, if x belongs to $S_{sk}(p)$, then, the output of x may not always be computable at specialization time for all values of `oref(`p`)`.

Because of these heuristics, the skeleton that is built by this algorithm is suboptimal. Later in this section, we will show using the example shown in Figure 5.4 how the algorithm selects suboptimal skeletons because of the several simplifying heuristics used by the algorithm.

The skeleton-building algorithm that handles cyclic SSA dataflow graphs (presented in Section 5.2) is also designed around these decisions.

5.1.3. The skeleton inheritance heuristic

In this section, we point out a source of high runtime costs in `BuildAcyclicScopeSkeletons` and present a simplifying, but suboptimal, heuristic that leads to reduced costs.

Note that `BuildAcyclicScopeSkeletons` builds a new skeleton at *every* invariant object reference node (Line A in Figure 5.2). After converting these skeletons to scopes, a cost-benefit analysis selects the most profitable set of non-overlapping scopes. However, this process can be expensive because a method typically has many object reference nodes. The number of skeletons built by the algorithm affects the scope-building cost in two ways:

- Cost of building the skeletons, and

- Cost of identifying a profitable set of non-overlapping scopes. More the scopes, more the choices that the cost-benefit analysis has to analyze.

This cost can be reduced by building a new skeleton only when necessary rather than build a new skeleton at every invariant object reference node. Consider the example shown in Figure 5.4. The figure shows the SSA dataflow graph for an expression from the `FindTreeNode` method in

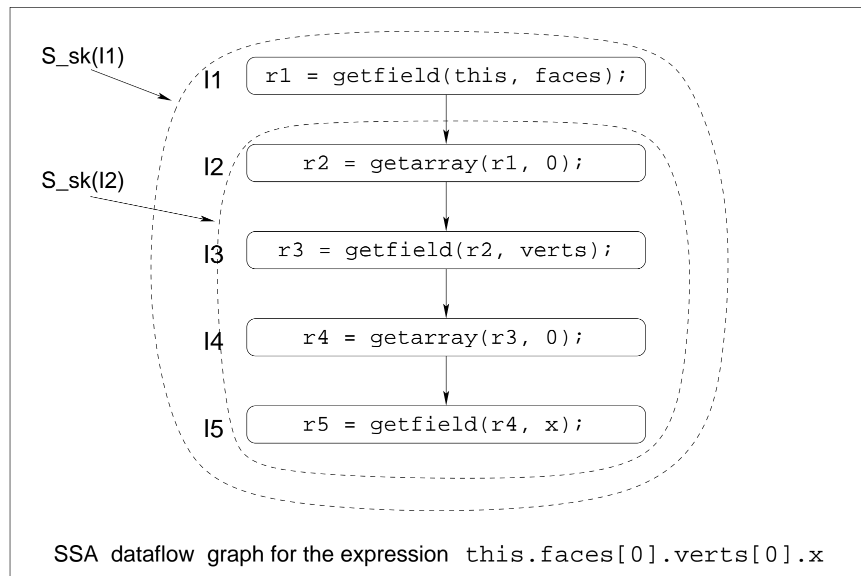

Figure 5.4. Example showing two nested skeletons where the outer skeleton subsumes the inner one

`raytrace`. The SSA graph for this expression contains five chained object reference nodes. Since we know that the octtree is invariant, all these nodes also happen to be invariant object reference nodes. Therefore, all of them are candidates for building scope skeletons. The figure shows two of the skeletons, $S_{sk}(I_1)$ and $S_{sk}(I_2)$. As shown in the figure, the inner skeleton is fully contained within the outer, i.e. $S_{sk}(I_2)$ is a subgraph of $S_{sk}(I_1)$. This is true in general, not just for this particular example. Given a SSA dataflow edge $u \rightarrow v$ where both u and v are invariant object reference nodes, the scope skeleton $S_{sk}(v)$ will be a subgraph of $S_{sk}(u)$ (put another way, $S_{sk}(v)$ will be completely nested in $S_{sk}(u)$).

Let us understand the implications of this nesting property for specialization. Refer back to

the example in Figure 5.4. For this example, the scopes happen to span the same instructions as their corresponding skeletons. Specializing the inner scope will eliminate instructions I_2 through I_5. Specializing the outer scope will eliminate instructions I_1 through I_5. However, this does not necessarily imply that the outer scope is a better specialization scope than the inner one. If the object reference profile shows that there are 500 frequently seen values of $this$ and only 5 frequently seen values of r_1 (due to data structure sharing), it is beneficial to select the inner scope over the outer scope. However, if there are 20 frequently seen values of $this$ and 20 frequently seen values of r_1, it is clearly beneficial to select the outer scope over the inner scope.[3]

This scenario can be recast into the following skeleton-building question: *given a SSA dataflow edge $u \rightarrow v$ where both u and v are invariant object reference nodes, and u belongs to a skeleton, should a new skeleton be built at v?* One answer to this question has been given in `BuildAcyclicScopeSkeletons`. It always builds a new skeleton at v. The cost-benefit analysis step shown in Figure 5.1 selects the best set of scopes in the end. However, besides being a high-overhead approach, in some cases, it is not necessary to build a new skeleton at v.

In this dissertation, we use a simplifying, but suboptimal, heuristic to answer the above question. Given a SSA dataflow edge $u \rightarrow v$ where both u and v are invariant object reference nodes, a scope skeleton is never built for v – it inherits the scope of u. However, this *skeleton inheritance heuristic* is suboptimal, because for the earlier example (Figure 5.4), this heuristic will essentially always select the outer scope over the inner scope.

We now briefly examine the implications of the skeleton inheritance heuristic. Given an invariant object reference node u and its skeleton $S_{sk}(u)$, skeletons are never built for the nodes in $S_{sk}(u)$ except for u. In combination with the greedy strategy that always builds maximal skeletons, this heuristic minimizes the number of skeletons built by the algorithm which reduces the runtime cost

[3]Note that the third scenario: 5 values of $this$ and 500 values of r_1 is not possible. The number of unique values of $r_1 = this.faces$ cannot be greater than the number of unique values of $this$ because $this.faces$ is invariant.

of skeleton building.

Thus, in the course of this discussion, we have presented two answers to the skeleton-building question: always-build vs. never-build. The always-build answer can ultimately lead to the “best” scopes, but has high runtime execution costs. In contrast, the never-build answer can be very efficiently implemented at runtime, but can lead to suboptimal scopes. Clearly, these two answers represent two extremes of a spectrum. Exploring other alternatives and improving upon the suboptimal skeleton inheritance heuristic studied in this dissertation is left for future work.

5.2. Building scope skeletons for arbitrary SSA graphs

In this section, we present an algorithm that computes scope skeletons for arbitrary SSA graphs. We first present a definition of scope skeletons that is valid for both cyclic as well as acyclic SSA graphs. Then, we present the skeleton-building algorithm formulated as a dataflow analysis on the SSA graph.

5.2.1. Defining a scope skeleton for arbitrary SSA graphs

In this section, we first describe the properties of a scope skeleton for cyclic SSA graphs using examples to clarify the definition.

Figure 5.5 shows a code snippet with a loop and the corresponding SSA dataflow graph. The figure also shows a scope skeleton for the getarray instruction. The skeleton includes the two dataflow loops shown in the figure. The intent behind including dataflow loops within a skeleton is to allow unrolling of the control-flow loops that compute the values of the dataflow loops. Therefore, in the example, the scope corresponding to the skeleton would include the entire loop which will be unrolled during specialization.

The cyclic skeleton shown in Figure 5.5 does not meet the definition of a skeleton defined in Section 5.1.1. First, the skeleton is no longer a forward slice seeded at the getarray instruction (the

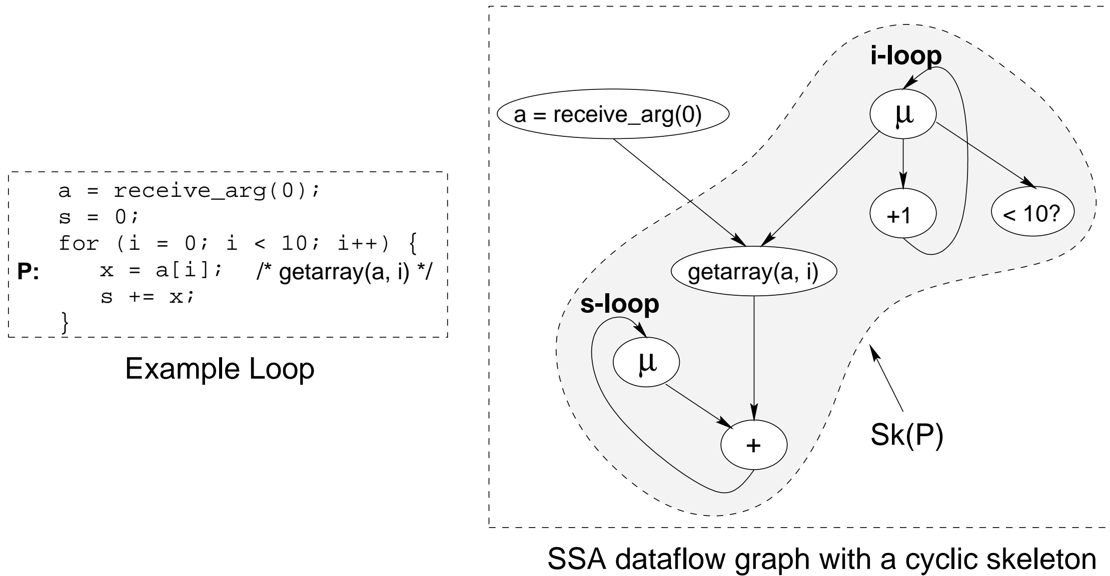

Figure 5.5. Example showing a skeleton in a cyclic SSA graph. The high level source code contains a loop which leads to the two data-flow loops shown in the SSA graph. Since the SSA graph is in canonical form, the constant assignments `i=0`, and `s=0` are folded into the μ-nodes

skeleton root). The skeleton shown in the figure includes the backward slice of the getarray instruction along the dataflow edge that provides the array index. Second, it is unclear what invariance of the getarray node means because the array index comes from i-loop (a dataflow loop) which is not a constant.

We now provide a revised definition of a scope skeleton that generalizes the acyclic skeleton definition. Let p be an object reference node (not necessarily invariant). Then, $S_{sk}(p)$, a scope skeleton seeded at p is a *forest of connected subgraphs* of the SSA graph that includes p. If a node x is in $S_{sk}(p)$, then x satisfies one of the following conditions:

1. x is an invariant object access node and `oref(`x`) = oref(`p`)`.

2. x is an invariant object reference node, and all its input nodes are in $S_{sk}(p)$.

3. x is a ϕ-node, all its input nodes are in $S_{sk}(p)$ and compute the same value along all incoming edges.

4. x is a non-ϕ scalar computation node, and all its input nodes are in $S_{sk}(p)$.

5. x represents `getarray(a, i)`, `i` is computed by a dataflow loop (called `i-loop`), and all the following conditions are satisfied:

   (a) Let d be the instruction that defines `a`. Then, either p or d control-dominates all the nodes in `i-loop`.

   (b) The invariance of some arbitrary element of the array can be verified using the store profile.

Note that the skeleton shown in Figure 5.5 satisfies the above definition (assuming that the array `a` is completely invariant). Except for the root instruction (`getarray(a, i)`), all other instructions in their skeleton have their input nodes in the skeleton. For the `getarray(a, i)` node, the definition node of `a` (`a = receive_arg(0)`) control-dominates all the nodes of `i-loop`.

We now discuss the need for these conditions in greater detail. Conditions 1–4 are similar to those presented for an acyclic skeleton in Section 5.1.1. Condition 5 is new. Let us understand why it has been included. If x, a `getarray(a, i)` node, receives its input from a dataflow loop, this implies that x accesses multiple array elements within loop(s). Therefore, if the node is included in the scope skeleton, the loops can be unrolled. However, an unrolled array access can be specialized *only if*: (1) the unrolled array values can be computed at specialization time, (2) the unrolled index values can be computed at specialization time, and (3) the unrolled array accesses are invariant. Conditions 5a ensures that the first requirement is met. Conditions 1–4 ensure that the second requirement is met. Condition 5b ensures that the third requirement is met.

Condition 5a verifies that the array is either defined outside the loop (d control-dominates `i-loop`), or is computable from the specialization key values available at p (p control-dominates `i-loop`). The need for these checks is clarified by the examples in Figure 5.6 and Figure 5.7. In Figure 5.6, suppose we were trying to build a skeleton for the `use a[i]` node. In order to add

```
for (i = 0; i < 100; i++) {
   int[] a = GetRandomIntArray();
   ...  use a[i] ...
}
```

Figure 5.6. Example illustrating the utility of condition 5a in the definition a scope skeleton. In this example, the loop cannot be unrolled because the array is defined within the loop.

`a[i]` to the skeleton, conditions 5a and 5b have to be satisfied since `i` comes from a dataflow loop. However, since `a` is defined within the loop, the definition point does not control-dominate the loop. Neither does the skeleton root (`use a[i]`), control-dominate the loop. Therefore, `a[i]` cannot be added to the skeleton. It can be verified that this is the right decision since it is not beneficial to unroll and specialize the unrolled iterations; the value of the array is not known and hence, the array access `a[i]` cannot be eliminated. In Figure 5.7, suppose we were trying to build a skeleton

```
o = receive_arg(0);
x = o.f;          /* skeleton root */
for (i = 0; i < 10; i++) {
   foo(x);        /* foo could potentially modify x */
   int[] a = x.a;
   ...  use a[i] ...
}
```

Figure 5.7. Example illustrating the utility of condition 5a in the definition of a scope skeleton. In this example, the loop can be unrolled even though the array is defined within the loop.

for the `x = o.f` node. Let us assume that the access `x.a` is invariant. In this example, `a[i]` can be added to the skeleton even though `a` is defined within the loop[4] because the skeleton root (`x = o.f`) control-dominates the loop. It can be verified that this is the right decision because the loop can be unrolled and specialized. The value of `a` is known for all the unrolled iterations because the `x.a` access is invariant. Therefore, the array accesses `a[i]` can be eliminated (assuming that they are invariant).

Condition 5b is a heuristic that generalizes the (in)variance of all array elements referenced by x

[4]In this example, the "loop-invariant" expression `x.a` has been left alone because of the intervening call to `foo`.

based on the (in)variance of a single arbitrary array element. This heuristic is similar in spirit to the hot-object approximation heuristic.

5.2.2. Algorithm `BuildScopeSkeletons`

Unfortunately, unlike the acyclic scope skeleton definition, the definition for a general scope skeleton is not constructive. From the definition, it is not directly clear how to identify a forest of connected subgraphs starting at the skeleton root that satisfies the conditions 1–5. Satisfying conditions 1–4 is also not straightforward in the presence of loops because these conditions imply a topological order of processing the nodes which is not possible for cyclic graphs.

In this section, we present a skeleton-building algorithm that is formulated as a dataflow analysis. We first present the heuristics that are central to this algorithm. We then present an overview of the algorithm and then present the details of the dataflow analysis.

Background: Simplifying heuristics

Algorithm `BuildScopeSkeletons` is built on various ideas that have been explored so far. Three heuristics (that we presented in the course of acyclic skeleton building) are integral to the algorithm and these are summarized below:

- *Hot-object approximation heuristic:* At all object reference nodes, the algorithm uses the hot object as a representative object to decide the (in)variance of the object reference node.

- *Building maximal scope skeletons:* The algorithm greedily grows a scope skeleton till it can no longer be grown.

- *Skeleton inheritance heuristic:* Given a SSA dataflow edge $u \rightarrow v$ where both u and v are object reference nodes, v inherits the skeleton of u, if u has one.

Overview of algorithm

In this section, we present an overview of the algorithm and the key ideas behind it. Just like the acyclic scope building algorithm presented in Section 5.1.2, this algorithm starts new skeletons at object reference nodes and grows them by processing all target nodes.

The basic algorithmic step involves processing a node v, in the context of its input nodes. While processing v, the algorithm employs all the three heuristics presented previously. The specific details of processing v depends on the nature of v and its input nodes. We present these details later on in this section. As an example, let us assume that v is an object reference node and has exactly one input node, u, which is also an object reference node. First, to determine if v is invariant, the algorithm uses the hot-object approximation heuristic. Second, if v is invariant, and u belongs to skeleton S, the algorithm uses the greedy strategy and expands S to include v without any cost-benefit analysis. Third, the algorithm uses the skeleton inheritance heuristic to avoid building a new skeleton at v. In the second step above, before adding v to S, the algorithm verifies that the resulting skeleton satisfies the conditions in the skeleton definition provided earlier. The specific checks depend on the nature of u.

Let us now understand how the algorithm handles dataflow loops and discovers those that can be unrolled and specialized. In SSA graphs, μ-nodes[5] signal the presence of dataflow loops. During dataflow analysis, if the μ-node receives constant (but, different) values along its two input edges, the algorithm assumes that, if unrolled, the computed values will all be constant. At the output of such μ-nodes, the algorithm uses a special lattice value $\mathcal{LC}$ to capture the loop-constant abstract value. By propagating $\mathcal{LC}$ through the SSA graph during dataflow analysis, the algorithm abstractly propagates the information that the unrolled loop will produce a sequence of constant values.

This process is pictorially shown in Figure 5.8. Figure 5.8 shows the example from Figure 5.5

[5] Recall that μ-nodes are ϕ-nodes in loop headers.

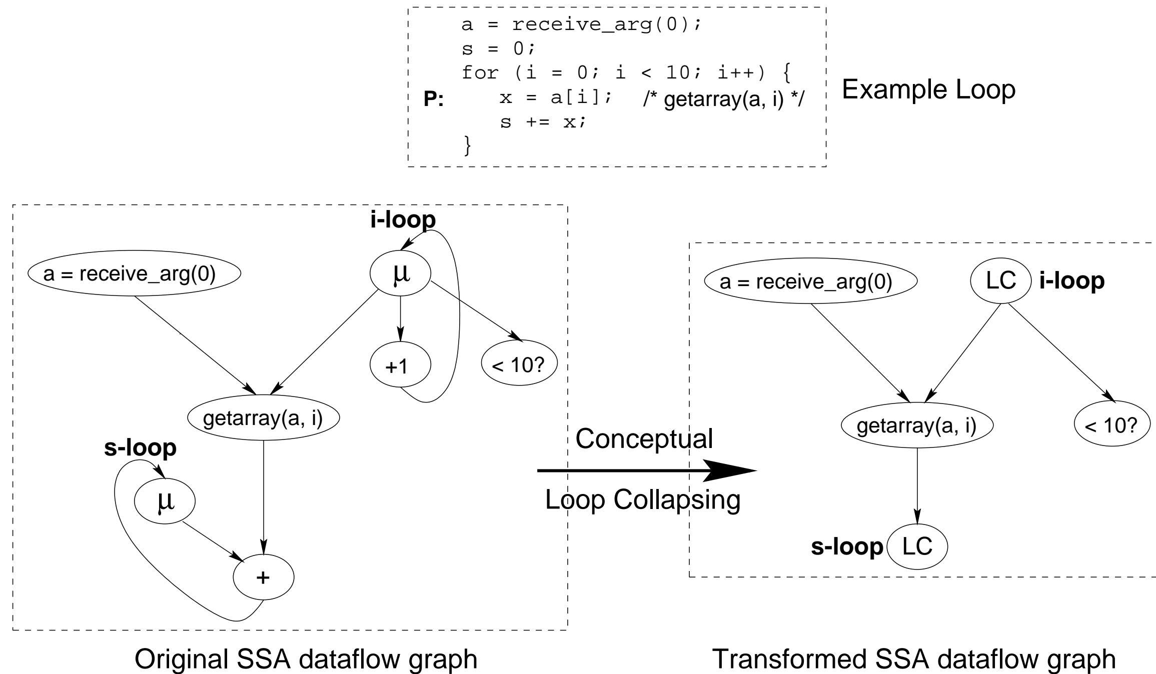

Figure 5.8. Conceptual loop transformation that is implemented by the algorithm to collapse dataflow loops that compute a sequence of constants. In the transformed SSA graph, *LC* represents a loop-constant abstract value.

and a conceptual loop collapsing step which collapses dataflow loops and transforms the cyclic SSA graph into an acyclic one. The abstract loop-constant value captures the information that the loop will be unrolled and that the values computed after unrolling will be constant. After this conceptual collapsing, when the `getarray` node is processed, the algorithm recognizes that the `getarray` is accessing a sequence of array elements with constant indices. It verifies that the `getarray` node satisfies conditions 5a and 5b in the skeleton definition and adds the node to a new skeleton. In the process, it conceptually adds the constant loop index (i-loop LC node) to the skeleton which marks the corresponding control-flow loop for unrolling.

Having seen an overview of the algorithm, we now turn our attention to the details of the dataflow analysis that incorporates all the ideas discussed in this section.

Skeleton Lattice

The skeleton building algorithm (implemented as a dataflow analysis) uses the dataflow lattice shown in Table 5.1. During dataflow analysis, instructions are associated with values from the `SSLattice` lattice.

$L_1 : \top \rightarrow Skeleton \rightarrow \bot$ $Skeleton$ denotes all possible skeletons	$L_2 : \top \rightarrow c \rightarrow \bot$ c denotes all possible constants and includes $\mathcal{LC}$, a special constant which denotes a sequence of constants computed by an instruction in a loop nest.
$S \sqcap \top = S;$ $\top \sqcap S = S;$ $S_1 \sqcap S_2 = S_1\ if\ (S_1 = S_2);$ $S_1 \sqcap S_2 = \bot;\ if\ (S_1 \neq S_2);$	$c \sqcap \top = c;$ $\top \sqcap c = c;$ $c_1 \sqcap c_2 = c_1\ if\ (c_1 = c_2);$ $c_1 \sqcap c_2 = \bot;\ if\ (c_1 \neq c_2);$
`SSLattice`: $L_1 X L_2$ $(S_1, c_1) \sqcap (S_2, c_2) = (S_1 \sqcap S_2, c_1 \sqcap c_2);$	

Table 5.1. Lattice used by the dataflow analysis algorithm that computes scope skeletons on a SSA graph

The L_1 lattice is used to record skeletons and propagate them along SSA edges during dataflow analysis. If a node has $\top$ as its L_1 lattice value, this means that the node does not yet belong to any skeleton. If a node has $\bot$ as its L_1 lattice value, this means that the node cannot be part of any skeleton. Let us now understand the intuition behind the $\sqcap$ operator. In our specialization model, since scopes are restricted to single-variable specialization keys, whenever a multi-input instruction x (like ϕ, μ, OP, `getarray`) receives non-identical skeletons along its input edges, neither skeleton can be expanded to include x because the output of x can be computed only when values from all its input skeletons are available. Therefore, $S_1 \sqcap S_2$ is $\bot$ if $S_1 \neq S_2$. In a specialization model that can handle multi-variable keys, potentially, a new merged scope can be created in which case the $\sqcap$ operator will be different from the one shown in Table 5.1.

The L_2 lattice captures the nature of value flow along dataflow edges and it is present for two reasons: (1) for `getarray` instructions, it enables the array index to be discovered (2) it also enables

unrollable dataflow loops to be discovered – this is accomplished using the special lattice value $\mathcal{LC}$ as discussed in the algorithm overview. The details will become clearer when the transfer functions are explained.

Dataflow Analysis: Preliminaries

```
class Skeleton {
   Node    root;       /* The skeleton root */
   Set     instrs;     /* Instrs that belong to this skeleton */
   Set     loops;      /* Loops that have be unrolled */
   SSA-reg lookupReg;  /* Key for the enclosing scope – oref(root) */
   Instr   lookupPt;   /* Lookup point for the enclosing scope */
}

Skeleton Set BuildScopeSkeletons(Method m)
{
   Let sp          = Store Profile;
   Let oap         = Object-Access Profile;
   Let cfg         = m.cfg;
   Let ssa         = m.ssa;
   Let slMap       = new HashMap; /* SSA-Reg → SSLattice */
   Let skeletonMap = new HashMap; /* SSA-Reg → Skeleton */
   Let loopMap     = new HashMap; /* SSA-Reg → Loop Set */
   Initialize: slMap(r) = (⊤,⊤) for all SSA registers r;

   Perform a dataflow analysis on the SSA graph using SSLattice and the
   transfer functions shown in Table 5.2.

   return GetSkeletons(slMap, loopMap); /* Figure 5.15 */
}

Comments
   slMap(r:SSA-Reg) returns the SSLattice value for r;
   slMap(c:Constant) returns (⊤, c);
   skeletonMap(r:SSA-Reg) returns the skeleton with lookup key r;
```

Figure 5.9. BuildScopeSkeletons routine

Figure 5.9 shows the scope building algorithm as a dataflow analysis. For every instruction in the SSA graph, the analysis essentially computes the skeleton it belongs to. After the analysis completes,

the set of valid skeletons are recovered from the dataflow facts (GetSkeletons routine).

A skeleton is represented using the class definition shown in the figure. The first three fields of this class are self-evident. The second two fields of this representation correspond to the scope enclosing the skeleton. Given a skeleton $S_{sk}(p)$, the lookup key for the enclosing scope will be oref(p) where p is the skeleton root, root. For many skeletons and scopes, the lookup point will be identical to the skeleton root, root. However, for some cyclic loops, the scope lookup point will be in a loop pre-header whereas the root will be within the loop. This is true for the example shown in Figure 5.5 where the scope lookup point is outside the loop, whereas the skeleton root is the getarray instruction (x = a[i]).

The algorithm uses slMap to keep track of dataflow facts associated with a SSA node. This map associates dataflow information with registers because the dataflow analysis processes only those instructions that compute a result in a register. If a constant c is used to lookup slMap, it returns $(\top, c)$. This trick eliminates many checks in the algorithm and simplifies its presentation.

The algorithm uses skeletonMap to keep track of the lookup keys used by the enclosing scopes of the skeletons identified by the algorithm. This map is used to merge skeletons whose enclosing scopes use the same lookup key. We will illustrate the importance of skeleton merging using an example later on in this chapter.

The algorithm uses loopMap to keep track of the loops that are unrolled by a skeleton. These loops are associated with instructions that induce unrolling (getarray or getfield instruction). Since instructions are uniquely associated with their destination registers, loopMap uses the destination register of an instruction to track unrolleable loops.

Transfer functions

The transfer functions for various SSA nodes are shown in Table 5.2, Figure 5.11 and Figure 5.12. It can be verified that all transfer functions maintain the invariant that if either element of the SS-

Instruction	Transfer Function
$\mathtt{r} = \mathtt{recv_arg}(...);$ $\mathtt{r} = \mathtt{call}(...);$ $\mathtt{r} = \mathtt{new}(...);$	$\mathtt{slMap}(\mathtt{r}) = (\bot, \bot);$ $\mathtt{slMap}(\mathtt{r}) = (\bot, \bot);$ $\mathtt{slMap}(\mathtt{r}) = (\bot, \bot);$
$\mathtt{r} = \mathtt{OP}(\mathtt{r}_1, \mathtt{r}_2)$	$(\mathtt{S}_1, \mathtt{v}_1) = \mathtt{slMap}(\mathtt{r}_1);$ $(\mathtt{S}_2, \mathtt{v}_2) = \mathtt{slMap}(\mathtt{r}_2);$ $\mathtt{S} = \mathtt{S}_1 \sqcap \mathtt{S}_2;$ $\mathtt{v} = \mathtt{OP}(\mathtt{v}_1, \mathtt{v}_2);$ $\mathtt{slMap}(\mathtt{r}) = ((\mathtt{S} = \bot) \| (\mathtt{v} = \bot) \; ? \; (\bot, \bot) : \; (\mathtt{S}, \mathtt{v});$
$\mathtt{r} = \phi(\mathtt{r}_1, ..., \mathtt{r}_k)$	$(\mathtt{S}, \mathtt{v}) = \mathtt{slMap}(\mathtt{r}_1) \sqcap ... \sqcap \mathtt{slMap}(\mathtt{r}_k);$ $\mathtt{slMap}(\mathtt{r}) = ((\mathtt{S} = \bot) \| (\mathtt{v} = \bot)) \; ? \; (\bot, \bot) : \; (\mathtt{S}, \mathtt{v});$
$\mathtt{r} = \mu(\mathtt{r}_e, \mathtt{r}_b)$	$(\mathtt{S}_e, \mathtt{v}_e) = \mathtt{slMap}(\mathtt{r}_e);$ $(\mathtt{S}_b, \mathtt{v}_b) = \mathtt{slMap}(\mathtt{r}_b);$ $\mathtt{S} = \mathtt{S}_e \sqcap \mathtt{S}_b;$ if $((\mathtt{S} = \bot) \| (\mathtt{v}_e = \bot) \| (\mathtt{v}_b = \bot))$ $\mathtt{slMap}(\mathtt{r}) = (\bot, \bot);$ elseif $(\mathtt{v}_b = \top)$ $\mathtt{slMap}(\mathtt{r}) = (\mathtt{S}, \mathtt{v}_e);$ elseif $(\mathtt{v}_e = \top)$ $\mathtt{slMap}(\mathtt{r}) = (\mathtt{S}, \mathtt{v}_b);$ elseif $(\mathtt{v}_e = \mathtt{v}_b)$ $\mathtt{slMap}(\mathtt{r}) = (\mathtt{S}, \mathtt{v}_e);$ else $\mathtt{slMap}(\mathtt{r}) = (\mathtt{S}, \mathcal{LC});$
$\mathtt{r} = \mathtt{getstatic}(\mathtt{f});$	$\mathtt{slMap}(\mathtt{r}) = \mathtt{sp.IsInvariant}(\mathtt{f})$ $? \; (\top, \mathtt{StaticValue}(\mathtt{f})) : \; (\bot, \bot);$
$\mathtt{r} = \mathtt{arraylen}(\mathtt{a})$ $\mathtt{r} = \mathtt{getfield}(\mathtt{o}, \mathtt{f})$ $\mathtt{r} = \mathtt{getarray}(\mathtt{a}, \mathtt{i})$	$\mathtt{slMap}(\mathtt{r}) = \mathtt{ProcessArrayLength}(\mathtt{I});$ $\mathtt{slMap}(\mathtt{r}) = \mathtt{ProcessGetField}(\mathtt{I});$ $\mathtt{slMap}(\mathtt{r}) = \mathtt{ProcessGetArray}(\mathtt{I});$

Table 5.2. Transfer functions used by `BuildScopeSkeletons` to build the scope skeletons on the SSA graph

`Lattice` value is $\bot$, both elements are lowered to $\bot$.

We now provide an intuitive interpretation of the output of some of the transfer functions.

- If the output of a transfer function is $(\top, c)$, this means that the instruction does not belong to any skeleton, and computes the value c.

- The transfer function output for a `call` instruction is $(\bot, \bot)$. This means that the call does not belong to any skeleton, nor is the output of the call available for instructions that use the call result. Clearly, this transfer function can be improved in certain cases. For example, when

the callee is a pure function (ex: library mathematical functions), the input skeleton can be expanded to include the call, and the call result can be made available to target SSA nodes.

- Examining the transfer function for the ϕ-node, it can be verified that this implements condition 3 of the definition of a skeleton. Therefore, if a ϕ-node receives two different values along its branches, the ϕ-node is not added to any skeleton.

- Let us now examine the transfer function for a μ-node. A μ-node signals the presence of dataflow loops. The transfer function shows that if the inputs to the node are constant, but, non-identical values, the output goes to $\mathcal{LC}$. For example, the index computation `for (i = 0; i < 10; i++)` will have a μ-node of the form: $i_1 = \mu(0, i_2)$. It can be verified that this node will stabilize with the lattice value $(\top, \mathcal{LC})$. This captures the fact that this index computation is not part of any skeleton but computes a sequence of constant values. Note that this lattice value is similar to $(\top, c)$ for the scalar constant c.

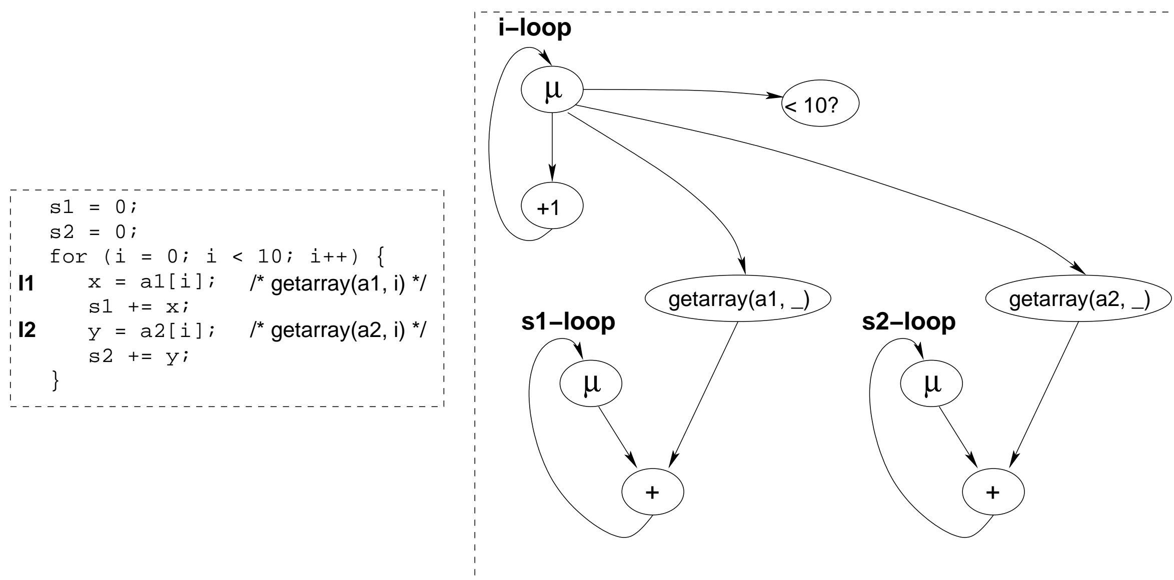

Figure 5.10. Example showing the importance of the $\top$-skeleton

Figure 5.10 shows an example which highlights the importance of $\top$ as a skeleton lattice value. The figure shows two separate arrays being accessed in the loop. For the i-loop μ-node, the

analysis will compute the lattice value $(\top, \mathcal{LC})$. By not including μ-node in any skeleton, it enables both getarray nodes to use it for unrolling the loop. As a result, the algorithm will identify two skeletons, one each for the two arrays. The cost-benefit analysis will then pick the better scope.

- For the `getstatic` node, if the accessed field is invariant, note that the transfer function does not assign it to any skeleton. This is because `getstatic` instructions do not have any input values – therefore, as long as the content of the static field is invariant, its value can be used by any skeleton.

The algorithm focuses on object reference SSA nodes (`arraylength`, `getfield`, and `getarray`). New skeletons are built only at these instructions – this follows the definition of a skeleton whose roots are always object reference nodes. The transfer functions for `ProcessGetField` and `ProcessArrayLength` are shown in Figure 5.11, and the transfer function for `ProcessGetArray` is shown in Figure 5.12. We discuss these transfer functions in the following paragraphs.

Note that all these three transfer functions first check for the presence of a hot object. The object-access profile might not find any hot object for an instruction if the instruction is in an infrequently basic block. In such cases, the instruction is not processed any further.

Transfer function for `arraylength` and `getfield`: Since conditions 1, 2, and 5 of the definition of a skeleton requires all object reference nodes to be invariant (except `getarray` instructions whose index computation comes from a dataflow loop), the transfer function for `getfield` in Figure 5.11 first checks for invariance of the referenced field of the hot object. However, there is no such check in `ProcessArrayLength` because array lengths are immutable.

Since both `arraylength` and `getfield` are single-input object reference nodes, they inherit the skeletons of their source nodes if one exists. If not, they build a new skeleton as shown in the transfer functions.

```
SSLattice ProcessArrayLength(I: r = arraylen(a))
{
        /* 1. Check for existence of a hot array */
   a_hot = oap.GetHotObject(I);
   if (a_hot = null) return (⊥,⊥);

        /* 2. Build new skeleton or grow existing skeleton */
   (S, _) = slMap(a);
   if (S = ⊤) return (⊤,⊤);
   if (S = ⊥) S = BuildNewSkeleton(I, I, a); /* No existing skeleton */
   return (S, a_hot.length);
}

SSLattice ProcessGetField(I: r = getfield(o, f))
{
        /* 1. Check for invariance of the hot object */
   o_h = oap.GetHotObject(I);
   if ((o_h = null) || !sp.IsInvariant(o_h.f)) return (⊥,⊥);

        /* 2. Build new skeleton or grow existing skeleton */
   (S, v) = slMap(o);
   if (S = ⊤) return (⊤,⊤);
   if (S = ⊥) S = BuildNewSkeleton(I, I, o); /* No existing skeleton */
   else { /* Expand existing skeleton */
      Instr src = ssa.GetDefinition(o);
      if ((v = LC) && (src is a μ-node)) {
            /* o is defined by the μ-node which is in skeleton S. If I is in
            muLoop, o takes on new values in each iteration. Therefore, we
            decide to unroll muLoop. If I is outside muLoop, the value of
            o will be known only if S unrolls muLoop. If not, we have to
            build a new skeleton starting at I. */
         Let muLoop = src.InnerLoop;
         if (I.innerLoop = muLoop) loopMap.AddLoops(r, muLoop);
         else S = BuildNewSkeleton(I, I, o);
      }
   }
   return (S, o_h.f);
}
```

Figure 5.11. ProcessArrayLength, ProcessGetField transfer functions

```
SSLattice ProcessGetArray(I: r = getarray(a, i))
{
        /* 1. Check for existence of a hot array */
    a_hot = oap.GetHotObject(I);
    if (a_hot = null) return (⊥,⊥);

    (S_a, _) = slMap(a);
    (S_i, v_i) = slMap(i);
        /* 2. Index not computable */
    if ((v_i = ⊥) || (S_i = ⊥)) return (⊥,⊥);
    if ((v_i = ⊤) || (S_a = ⊤)) return (⊤,⊤);
        /* 3. Array and index come from different skeletons */
    if ((S_i ≠ ⊤) && (S_a ≠ S_i)) return (⊥,⊥);
        /* 4. Access of single array element */
    if (v_i ≠ LC) {
        if (!sp.IsInvariant(a_hot[v_i])) return (⊥,⊥);
        if (S_a = ⊥) S = BuildNewSkeleton(I, I, a);
        return (S, a_hot[v_i]);
    } else {
            /* 5. i is computed by a dataflow loop involving μ-node(s). Thus, I is access-
            ing multiple elements of the array a_hot. In order to specialize away these array
            accesses, we have to unroll all the loops involved in computing i. This is possi-
            ble only if: (1) I is present in the innermost loop, and (2) Condition 5a in the
            skeleton definition is satisfied */
        loops = GetLoops(i);
        ol = OutermostLoop(loops);
        il = InnermostLoop(loops);
        if (I.innerLoop ≠ il) return (⊥,⊥);
        if (S_a = ⊥) {
            Instr d = ssa.GetDefinition(a);
            if (!cfg.Dominates(d, ol.header)) return (⊥,⊥);
                /* If a is not part of an existing skeleton, the lookup point will be in the
                preheader of the outermost loop. */
            Instr lookupPt = ol.preheader.tail;
            S = BuildNewSkeleton(I, lookupPt, a);
        } else {
            S = S_a;
            if (!cfg.Dominates(S.lookupPt, ol.header)) return (⊥,⊥);
        }
        loopMap.AddLoops(r, loops);
        return (S, LC);
    }
}
```

Figure 5.12. `ProcessGetArray` transfer function

/* We want a new skeleton at `I` with lookup-reg `r`. If some other instruction has already built a skeleton (`S`) with this same lookup register, we can use it if (1) the `S.lookupPoint` dominates I, and (2) if `S` is still valid (During dataflow analysis, the lattice values keep changing. Therefore, the instruction that originally built `S` might have changed its lattice value.) */

```
Skeleton BuildNewSkeleton(Instr I, Instr lookupPt, SSA-Reg r)
{
   Skeleton S = skeletonMap.get(r);
   if((S ≠ null) && cfg.Dominates(S.lookupPt, lookupPt)){
      SSA-Reg d = S.root.destinationReg;
      (S_d, _) = slMap(d);
      if (S = S_d) return S;
   }
   S = new Skeleton();
   S.root        = I;
   S.lookupPoint = lookupPt;
   S.lookupReg   = r;
   skeletonMap.put(r, S);
   return S;
}
```

Optimizations:

1. If the lookup reg r comes from a memory-allocation instruction, then, there is no use building a new skeleton and the routine can return $\perp$.

2. If the object-access profile shows that r has "too many" (value decided empirically) values seen at runtime, the routine could return $\perp$ with the assumption that only a certain maximum number of specialized versions can be created for any scope.

Figure 5.13. `BuildNewSkeleton` routine

For `getfield`, the transfer function also checks if a loop can be unrolled. A loop can be unrolled if the getfield receives a value from a dataflow loop. The special lattice value $\mathcal{LC}$ signals this. The additional check (src is a μ-node) is an optimization – if the definition node is not a μ-node, it means that some other node would have dealt with the dataflow loop already (another `getfield` or `getarray` instruction).

Transfer function for `getarray`: The transfer function for the `getarray` instruction is the most complex of all – this is due to the fact that it is a two-input instruction and can induce unrolling of multiple nested loops.

- If the index is not computable, then the array element access cannot be specialized and therefore, the instruction cannot be added to any skeleton.
- If the index and the array come from non-identical skeletons, the instruction cannot be added to either skeleton because the specialization model only allows single-element specialization keys. However, if the index comes from a $\top$ skeleton, the index might be a scalar or a loop constant (Ex: Figure 5.8) and therefore the instruction can potentially be added to the skeleton.
- If the index is not $\mathcal{LC}$, the transfer function checks for the invariance of the array element being accessed. Condition 5b (for getarrays that access multiple array elements) is satisfied by this check as explained next. When a μ-node is processed for the first time, it passes along the value coming in from outside the loop (v_e) since the back-edge value (v_b) will be $\top$. Therefore, when a getarray is processed for the first time, it will *always* have a constant array index. Therefore, this dataflow analysis uses the (in)variance of $a[v_e]$ to generalize for all elements accessed by the instruction.
- If the index is $\mathcal{LC}$, the transfer function checks if condition 5a in the skeleton definition is satisfied. The specific clause that is checked depends on whether the array belongs to an

existing skeleton or not.

Building New Skeletons Having discussed the transfer functions for all instructions, we now discuss how new skeletons are built. Figure 5.13 shows the routine that builds new skeletons. The routine first checks whether the new skeleton can be merged with an already existing skeleton that has r as the lookup key. If there exists one (say S), it can be used if S satisfies some conditions. Firstly, S must control-dominate the lookup point requested for the new skeleton. Secondly, the S must still be valid. S might be invalid because the instruction that built S might have changed its lattice value (the skeleton component might have gone to $\perp$ or to a different skeleton). The check verifies that S is still valid by comparing it with the current lattice value of the instruction that built S. Figure 5.13 also shows a couple of self-explanatory optimizations that can be performed on this basic routine.

```
int s  = this.start; /* New skeleton S1 */
int cs = this.chars; /* New skeleton S2 */
int n  = cs.length;
for (i = s; i < n; i++) {
   char c = cs[i]; /* Skeleton conflict at the getarray node */
   ... use c ...
}
```

Figure 5.14. Example illustrating the importance of merging skeletons whose scopes use the same lookup key. The example shows that without merging, there will be unneeded skeleton conflicts.

Figure 5.14 shows why it is important to merge skeletons, where possible. When `this.start` is processed, the algorithm builds a new skeleton S_1 with lookup key `this`. When `this.chars` is processed, the algorithm requests another new skeleton. If the algorithm does not use S_1 and builds a new skeleton S_2 instead, there will be a skeleton conflict at the `cs[i]` getarray instruction. `cs` will be defined in S_2 whereas `i` will be defined in S_1. As a result, the analysis will fail to unroll the loop in the figure. However, by merging S_2 with S_1, this problem is avoided. In general, skeleton merging leads to a forest of connected subgraphs (rather than a single connected subgraph) that are

```
Skeleton Set GetSkeletons(HashMap slMap, HashMap loopMap)
{
        /* Add unrolleable loops to appropriate skeletons */
    foreach ((r, loops) ∈ loopMap) {
        (S_r, _) = slMap(r);
        if (S_r ≠ ⊥) add loops to S_r.loops;
    }

        /* Identify all valid skeletons with their instructions */
    Skeleton Set SS = {};
    foreach (entry (r, (S_r, _)) in slMap) {
        if (S_r ≠ ⊥) {
            Instr i = ssa.GetDefiningInstr(r);
            add i to S_r.instrs;
            add S_r to SS;
        }
    }

    return SS;
}
```

Figure 5.15. `GetSkeletons` routine

part of the scope skeleton. For example, the `FindTreeNode` method shown in Figure 1.1 leads to such a skeleton starting at the getfield expression `this.faces` with `this` being the lookup key.

Recovering the set of valid skeletons

After the dataflow analysis terminates, the set of valid skeletons are recovered from the dataflow facts recorded in `slMap` using the `GetSkeletons` routine shown in Figure 5.15. This routine is straightforward and the figure is self-explanatory.

5.3. Identifying scopes from the skeleton set

In this section, we show how to build specialization scopes from the set of skeletons that were identified by the dataflow analysis presented in the previous section. In this section, we also present

the routine that compute the costs and benefits from creating specialized versions for a scope.

Instruction	**Benefit**	**Remarks**
`getfield`	2.0	if loaded value is not float/double
	0.5	if loaded value is float/double
`getstatic`	2.0	if loaded value is not float/double
	0.5	if loaded value is float/double
`getarray`	3.0	if loaded value is not float/double
	1.0	if loaded value is float/double
`arraylength`	2.0	
`x = y`	0.5	many copies get eliminated
`mul/div`	10.0	for float/double operands
	5.0	for int/long operands
`add/sub`	2.0	for long operand
	1.0	for other operands
everything else	1.0	

Table 5.3. Table showing the values used by the cost-benefit analysis to estimate benefit of a scope

Figure 5.16 shows the code that builds the scope, and determines the number of specialized versions to create for the scope. The first step is to build a single-entry, multiple-exit CFG region. To do this, for every instruction $i \in sk.instrs$, the algorithm adds the basic block of i to the scope. More basic blocks are added to this initial set to grow the scope such that the single-entry, multiple-exit property is satisfied.

In order to determine the number of specialized versions to create for the scope, the algorithm first determines the cost and benefit of specializing the scope. In this dissertation, the cost-benefit model used is rudimentary and is presented here for the sake of completeness. It is shown in Figure 5.16, Figure 5.17, and Table 5.3.

The cost-benefit model presented here can be improved to better predict the benefits of specialization. It does not account for compilation costs very well – as a result, the specializer causes signification slowdowns for certain programs (presented in Chapter 9). In addition, it does not have a good model to account for i-cache effects due to code bloat. However, a more sophisticated spe-

```
class Scope {
   Set     bbs;           /* Basic blocks of this scope */
   Set     loops;         /* Loops that have be unrolled */
   SSA-reg lookupReg;     /* Key for the enclosing scope – oref(root) */
   Instr   lookupPt;      /* Lookup point for the enclosing scope */
   int     numVersions;   /* Number of versions of this scope */
   float   profit;        /* Profit from specializing this scope */
}

Scope IdentifyScope(Skeleton sk)
{
      /* Expand the scope such that the scope encloses loops in their entirety. Once
      that is done, the scope will satisfy the single-entry, multiple-exit property */
   Scope s = new Scope(sk); /* sets loops, lookupReg, lookupPt */
   s.bbs = AddBasicBlocks(sk.instrs);
   MakeSingleEntryMultipleExitRegion(s);

      /* Compute benefit */
   float benefit = 0;
   foreach (i ∈ sk.instrs) {
      float x = GetBenefit(i); /* uses benefit values shown in Table 5.3 */
      float lf = (i.innerLoop ∈ s.loops) ?  5.0 :  1.0;
      benefit += (x * lf);
   }

      /* Compute incremental cost of creating a specialized version; Combine two fac-
      tors: cost due to loop specialization, and cost due to regular specialization. */
   float f  = |sk.instrs|/totalInstrs; /* fraction of instrs. eliminated */
   float f1 = |s.bbs|*(1-f)*0.05;        /* code bloat factor */
   float f2 = |s.loops|*(1-f)*2.5;       /* code bloat factor due to loops */
   float f3 = f1*((f2 == 0)?1.0:f2);     /* overall code bloat factor */

      /* Determine # of specialized versions to create */
   float dispCost = 8; /* dispatch cost – assumes a pseudo-method dispatch */
   float incrCost = f3;
   Instr root = sk.root;
      /* Figure 5.17 */
   (n, f) = GetNumVersions(root, benefit, dispCost, incrCost);
   s.numVersions = n;
   s.profit      = benefit*f - dispCost - incrCost*n;

   return s;
}
```

Figure 5.16. Building a scope for a skeleton

```
(int,float) GetNumVersions(Instr root, float benefit,
                           float dispCost, float incr)
{
     /* Get number of versions for 97% profile coverage */
  int   n    = oap.Get97PercCoverage(root);
  float totX = 0.0;
  float oldX = 0.1;
  float oldcb = -∞;
  for (i = 0; i < n; i++) {
       /* x = contribution of spec. version i to total execution of the scope */
    float x = oap.GetCoverageOfVersion(i);
    float ratio = oldX/x; /* used to detect sharp knee in curve */
    float cb = benefit*(totX + x) - dispCost - incr*(1+i);
    float diff = cb - oldcb;
    if ((diff < 0.05) || ((ratio > 10) && (diff < 1)))
      break;
    totX += x; oldcb = cb; oldX = x;
  }
  return (i, totX);
}
```

Figure 5.17. Determining number of specialized versions for a scope

cializer can minimize slowdowns by using a better model that accounts for i-cache effects.

We did not do a very thorough job of researching and developing a good cost-benefit model in this dissertation because in the absence of a true dynamic optimization environment, the parameters and fine tuning of the cost-benefit model will be useless. The specific details and parameters of the model will be closely tied to the characteristics of the specialization system – the cost of compilation, the size of the i-caches, the cost of a memory access, and several other empirical measurements.

Even though we use a rudimentary cost-benefit model, there is one point worth mentioning. In determining the number of specialized versions to create, the routine `GetNumVersions` tries to detect a sharp knee in the curve of profile coverage vs. the number of specialized versions. For example, if the object-access profile shows that the object access frequencies at the lookup point are: 0.45, 0.02, 0.01, 0.005, 0.003, then the algorithm decides to create just a single specialized version. This technique proved to be quite useful in limiting the number of generated specialized versions.

5.4. Identifying a set of non-overlapping scopes

The final step of the automatic scope-selection algorithm is to identify the “best” set of non-overlapping and non-nesting scopes. This step uses the results of cost-benefit analysis in selecting scopes.

In this dissertation, a very ad hoc mechanism was used to implement this step. However, a more systematic way would be to use graph-coloring to solve this problem as follows. Firstly, an interference graph is created where scopes are nodes and there is an edge between two nodes if the corresponding scopes overlap with each other. Secondly, a graph coloring algorithm is applied to this graph using a single color. The single color is used to capture the fact that all scopes must be completely non-overlapping. The algorithm then selects those scopes that have been colored and rejects those that have not been colored.

CHAPTER 6

CREATING SPECIALIZED VERSIONS

In this chapter, we present an algorithm to create specialized versions of the scopes identified by the scope building algorithm. Given a scope `S(k)`, the specialization algorithm creates multiple specialized versions of the scope `S`, each of which is specialized with respect to:

- the (hot) value v_i of its key `k`;
- the values of the invariant portion of the heap.

This is accomplished with a two-step algorithm shown in Figure 6.1. The first step in this algorithm is to transform the CFG of the candidate method as shown in Figure 3.2. At this point, the CFG resembles the graph shown in Figure 3.2, at the right. After this transformation, the second step is to run a SCCP-based specializer on the transformed CFG. This automatically creates the required specialized versions. During the specialization process, we collect the set of memory locations that were specialized away to guard them against future modifications. We first present the details of the CFG transformation process and then present the details of our SCCP-based specializer.

6.1. Creating clones of specialization scopes

In more detail, the CFG transformation involves the following steps:

- creating clones of S
- splicing the clones into the cfg
- updating the SSA form

The pseudo-code shown in Figure 6.2 implements the above three steps. Lines 1 and 2 incrementally update the SSA form. An alternative to incrementally updating the SSA form would be to rebuild

```
MemoryLocation Set SCCP_Specialize(Method m, Scope Set SS)
{
   TransformCFG(m, SS);  /* Figure 3.2, Figure 6.2 */
   return SCCP_Specialize_Loops(m, new SCCP_State());
}
```

Figure 6.1. Algorithm SCCP_Specialize

```
TransformCFG(Method m, Scope Set SS)
{
   CFG cfg = m.cfg;
   foreach (Scope S ∈ SS) {
         /* Refer to Figure 5.16 for a definition of a Scope */
      k = S.lookupReg;
      n = S.numVersions;
      for (i = 1 to n) {
         Si = Create a clone of S;
         Add k = vi at the entry of Si;
1.       Update SSA form – rename variables defined in Si to use new names;
      }
      Splice the clones S1..Sn in cfg;
      Update SSA form – for every variable defined in S and live at the
2.    exit of S, add φ-nodes in basic blocks that correspond to control-flow
      merges of the clones.
   }
}
```

Figure 6.2. The TransformCFG routine

the SSA from scratch after the CFG is transformed [1]

6.2. SCCP-based specialization of the transformed CFG

SCCP-based specialization is carried out using a modified version of the Wegman-Zadeck Sparse Conditional Constant Propagation (SCCP) [89] algorithm. Due to its sparsity and linear-time efficiency, SCCP is a good fit for run-time specialization. More importantly, SCCP processes instructions conditionally which makes it possible to exploit it as a simple, but, powerful specializer based

[1] In our implementation, the SSA form is built from scratch after the CFG is transformed.

on the *online partial evaluation* strategy [56]. By evaluating conditionals (some of which may be run-time constants), SCCP ignores an instruction until it has been discovered to be reachable. In essence, SCCP performs two distinct optimizations simultaneously: unreachable code elimination and constant propagation. This combination allows SCCP to discover more constants and also eliminate more unreachable code than either optimization performed separately.

This property confers the online partial evaluation ability to SCCP. Like an online partial evaluator, it uses actual values (as opposed to just a knowledge of a variable being constant) in making specialization decisions: knowledge of constant values of other variables, as well as knowledge about the unreachability of basic blocks. However, clearly, this is not a very powerful online partial evaluator since it does not unroll loops, nor does it separate control-flow paths. These code transformations enable greater specialization to be performed. Later in this chapter, we present an extension which allows SCCP to unroll loops.

We now present an example to illustrate the power of SCCP as a specializer.

```
int foo(int i)
{
   x = i;
   if (x == 2)
      y = 3;
   else
      y = read_input();
   z = y + 1;
   return z;
}
```

Figure 6.3. Example showing the power of SCCP as a specializer

Consider the example code in Figure 6.3. Suppose we know `i` is a constant and always takes the value 2. For this input, it can be verified that the method always returns 4. A specializer based on the offline partial evaluation strategy will use a binding time analysis (BTA) to annotate expressions and statements as *static* or *dynamic*. Since BTA does not use the actual values of the static input to

perform its analysis, it will not be able to infer that the branch `(x == 2)` is always true. Therefore it will not be able to infer that `y` is always 3 and will thus annotate the statement `z = y + 1;` as dynamic. Thus, an offline partial evaluator cannot specialize this method to the single statement `return 4`.[2]

In contrast, it can be verified that an online partial evaluator which uses the value of `i` during the analysis process will be able to specialize this method to the single statement `return 4`. An online partial evaluator can infer that the value of `x` is 2 and can use this information to evaluate the branch and compute the value of `y` to be 3 which leads the specializer to determine the value of `z` to be 4.

Let us now consider a typical (non-SCCP) constant propagation algorithm that does not use conditional processing and initialize it with the mapping `i = 2`. Even though it can infer that the branch `x = 2` is always true, it can be verified that this algorithm will assume that `y` is not a constant because on the true-arm of the branch, `y` is 3, and on the false-arm of the branch, `y` is $\perp$. Therefore, it will assume that `z` is not a constant. Thus, this constant propagator cannot effectively specialize this example.

Let us now consider the SCCP algorithm and initialize it with the mapping `i = 2`. It can be verified that SCCP will be able to infer that `z` is a constant with the value 4. SCCP can do this because it processes instructions conditionally and will never process the instruction `y = read_input()`. As a result, it only uses the value of `y` coming in from the statement `y = 3`. Thus, for this simple example, SCCP is as powerful as an online partial evaluator.

6.2.1. Eliminating invariant memory accesses

Having presented an example of the utility of SCCP as a good basis for designing a specializer, we now show how it can be extended to eliminate invariant memory accesses.

[2]However, more aggressive offline specializers will be able to do this specialization by separating the two control-flow paths in the figure and maintaining BTA state for the duplicated control-flow paths. This is an example of polyvariant specialization and is implemented by DyC, among other offline partial evaluators.

As shown in Figure 6.2, to create a specialized versions of a scope `S`, a `k = ` v_i instruction is added at the beginning of each clone of `S`. Recall that in our specialization model (Section 3.1), only object references can be keys. Hence, v_i will be an address of an object. So, when SCCP is invoked on the transformed CFG, SCCP will work over a lattice containing concrete memory addresses, not only integers.

In order to eliminate invariant memory accesses, we need to modify how SCCP evaluates loads (which in our setting are the `getfield`, `getarray`, and `arraylength`bytecodes). SCCP uses the `VisitInstr` routine [89] to process instructions. We extend the routine as shown in Figure 6.4.

```
       if (I is <r = load p>) {
           a = LatticeVal(p);
           if (a is not a constant) { /* ⊤ or ⊥ */
               LatticeVal(r) = a;
1.         } else if (sp.IsInvariant(a)) {
2.             v = LoadMemory(a);
3.             LatticeVal(r) = v;
4.             add (a, v) to GuardedLocations
           } else {
               LatticeVal(r) = ⊥;
           }
       }
```

Figure 6.4. Modification to the `VisitInstr` routine

The effect of the extension (lines 1–3) is that when the analysis encounters a load whose argument is an address a (a constant generated by some other instruction), the store profile, `sp`, is consulted to check if a is an invariant memory location. If the profile shows that a has not been written, a is considered invariant. Then the load is performed on the concrete heap, and the loaded value v is used as if it were a constant. This load is considered constant and is eventually removed. Since the memory-invariance detection is based on optimistic dynamic analysis, in line 4 the pair (a, v) is added to the set of memory locations that will be guarded with run-time checks. We also maintain the value v for the purpose of avoiding race conditions between the application and the specializer,

which runs in a separate thread (see Chapter 8). Store instructions are not specialized away (unless they become control-flow unreachable).

Interaction with relocating garbage collectors

The above extension to SCCP to eliminate invariant memory accesses replaces symbolic object references in the program with the actual memory addresses of the objects. However, this is a problem if the garbage collector relocates objects because this changes the memory addresses of objects. This can lead to incorrect program behavior unless all instances of old object addresses in the program are replaced with the new relocated addresses. One simple way to do this address fixup is to maintain a table of all instructions into which object addresses are hardwired during specialization. Whenever the garbage collector relocates objects, it can consult this table to replace the old address with the new address in all instructions with hardwired object addresses.

6.2.2. Specializing Loops

Let us refer to the specialization algorithm we just presented as `SPEC_Specialize_Basic`. This algorithm does duplicate code either in the form of loop unrolling or in the form of path duplication. However, the authors of DyC [47] note that loop unrolling and specialization was crucial to getting good speedups in their specialization system. In this dissertation, code duplication in the form of loop unrolling has been implemented and studied. In the following section, we present an extension to `SPEC_Specialize_Basic` which enables it to unroll loops and specialize them.

The loops to be specialized are identified by the scope-finding algorithm (Section 5.2). The specializer attempts to unroll and specialize only these candidate loops. The specializer treats the input from the scope-finding algorithm as hints and aborts the unrolling if it discovers that the unrolling process might not terminate.

In the rest of this section, we use the term *loop specialization* to refer to the optimization where

loop iterations are peeled/unrolled and specialized.

Overview: Main idea

The central idea in specializing entire loops (and loop nests) is to peel loop iterations and specialize them one at a time until the loop terminates. To do this, the algorithm creates a copy of the iteration, and specializes it upto the loop back edge. Before peeling further iterations of the loop, the algorithm verifies that the peeling process will terminate. Effectively, the algorithm examines the loop exit conditionals of the specialized iteration. Consider an exit condition branch b. If the branch has been resolved to stay within the loop, the specializer peels off another iteration and repeats the process. If the branch has been resolved to exit the loop, the peeling stops – and we would have unrolled and specialized the entire loop. However, if the branch direction cannot be resolved at specialization time, the specializer terminates the unrolling process (because the loop exit condition is no longer a constant) and generates loop code for the remainder of the loop. In this case, the specializer will have unrolled an initial portion of the loop.

Details

Having presented the central idea, let us now examine this optimization in further detail. SCCP optimistically assumes [27] that (i) an instruction computes a constant value, unless proved otherwise, and that (ii) a basic block is unreachable, unless proved otherwise. This makes it harder to transform the program in the middle of the analysis because the intermediate dataflow state might not be correct. For example, during the analysis, SCCP might assume that a branch condition is constant. However, at the end of the analysis, it might find that the branch condition is not constant. Therefore, it would be incorrect to transform the program to eliminate the unreachable arm of the branch before the analysis terminates. More generally, the intermediate dataflow state of SCCP cannot be used for performing any CFG transformations or optimizations.

One way to overcome this limitation would be to perform loop specialization in a second pass over the CFG after `SCCP_Specialize_Basic` completes. However, as the following (contrived) example will show, this requires each candidate loop nest to be specialized one at a time. For the following example, four passes are required to fully specialize this method.

```
int foo(Class0 p)
{
 1. int s1 = 0;
 2. int n1 = p.x;
 3. int[] a1 = p.ia1;
 4. for (i = 0; i < n1; i++)                    /* Loop 1 */
 5.    s1 += a1[i];

 6. int n2 = (s1 < 20) ?  p.n1 :  p.n2;  /* Stmt. A */

 7. int s2 = 0;
 8. int[] a2 = p.ia2;
 9. for (i = 0; i < n2; i++)                    /* Loop 2 */
10.    s2 += a2[i];

11. return s2;
}
```

Figure 6.5. Example to clarify the interactions between a loop specialization pass and `SCCP_Specialize_Basic`

Consider the code shown in Figure 6.5. Let us assume the following:

1. The entire method `foo` is a specialization scope.
2. `p` is the specialization key.
3. All fields of objects referenced by `p` are invariant.
4. The spec-scope algorithm marks both loops (1 & 2) as specializeable.

It can be verified that the specialization of **Loop 2** requires the value of `n2` which depends on the value of `s1`, which in turn, requires the specialization of **Loop 1**. Because of these dependencies, four passes are needed to fully specialize the method as described below.

1. `SCCP_Specialize_Basic`: This pass computes the value of `n1`.

2. This pass computes the value of `s1` by specializing **Loop 1**.

3. `SCCP_Specialize_Basic`: This pass computes the value of `n2` using the value of `s1` computed in pass 2.

4. This pass specializes **Loop 2**.

While this would work, clearly, this is too inefficient. We now show how loop specialization can be integrated with the SCCP algorithm. As noted earlier, the primary obstacle in this integration is the problem of incorrect intermediate SCCP dataflow state.

The problem of incorrect intermediate dataflow state can be addressed by imposing a specific order on instruction processing. Let us revisit the example in Figure 6.5 and see how an integrated algorithm might specialize the method. If we apply `SCCP_Specialize_Basic` to instructions 1–3 that appear before **Loop 1** (and control-dominate it), we can unroll and specialize **Loop 1** without having to process **Stmt A** and the instructions in **Loop 2**. After that, we can apply `SCCP_Specialize_Basic` to instructions 6–7 before processing `Loop 2`. After this, we can unroll and specialize **Loop 2**. Finally, we can apply `SCCP_Specialize_Basic` to instruction 11. Thus, for this example, we were able to integrate loop unrolling and specialization by processing the instructions of the method in a very specific order. This is the idea behind integrating loop specialization with SCCP.

Computing a suitable visit order: We now show how to compute a visit order suitable for our purposes. We first consider an acyclic CFG. If we process the control-flow and data-flow edges in such a manner that the associated basic blocks are *always* processed in topological sort order,[3] then it can be easily verified that when a target basic block `b` is visited, the SCCP dataflow state for all

[3]Any arbitrary depth-first visit order [6] suffices – we choose the topological sort order for reasons of clarity in explaining this algorithm. A topological sort order can be computed using a depth first search algorithm – and in this case, the topological sort order will be identical to the depth-first visit order.

of b's control-flow predecessors will be the correct final state. Therefore, the subgraph that has been already processed can be transformed before SCCP terminates.

We now show how to compute a suitable visit order for cyclic CFGs. One possibility is to compute a topological sort order by ignoring the back edges. Yet another possibility is to use a depth-first visit order. However, neither of these strategies is sufficient, as shown by the example in Figure 6.6. It can be verified that both the visit orders shown in Figure 6.6 satisfy the topological ordering prop-

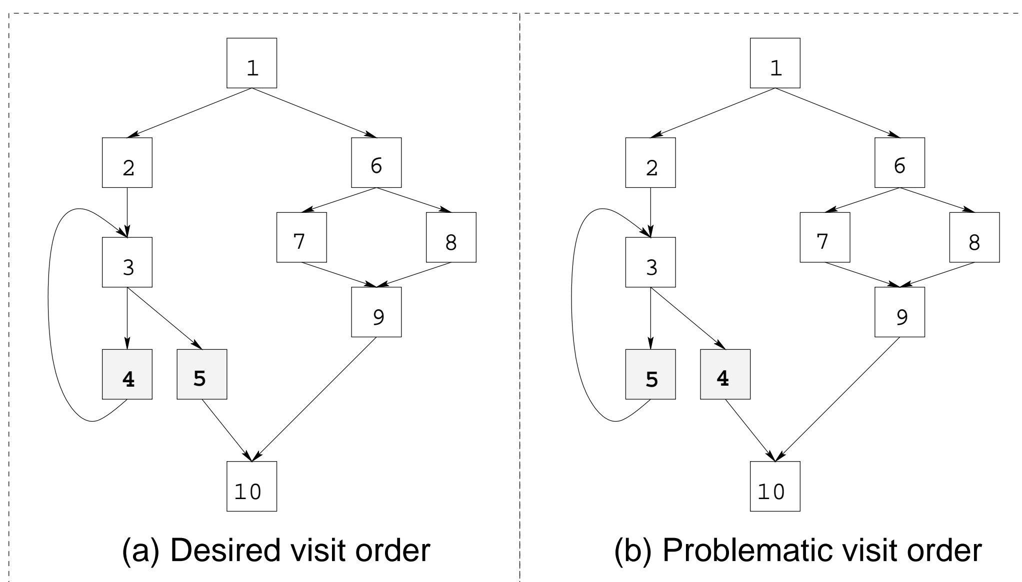

Figure 6.6. Example SCCP visit order

erty (on a graph where back edges have been removed) as well as the depth first ordering property. The only difference between the two visit orders is the position of the nodes with ids 4 and 5 as shown in bold in shaded nodes. However, only the visit order shown in Figure 6.6(a) is suitable for integrating loop specialization with SCCP. With the visit order shown on the right, the instructions in basic block 4 cannot exploit the results of unrolling and specializing the loop because it has been specialized even before the loop has been unrolled. Referring back to the example shown in Figure 6.5, a visit order similar to that in Figure 6.6(b) will fail to specialize Loop 2 because the value of n2 depends on the value of s1 which is not available till Loop 1 is specialized.

```
void ComputeBBVisitOrder(CFG c)
{
   List readyList = {};
   int visitID = 0;
   readyList.Add(c.entry);
   do {
         /* Assign visit id to b */
      BasicBlock b = readyList.GetHead();
      if (b.visitID == -1) b.visitID = visitID++;

         /* Add ready target nodes of b */
      Set dsts = b.TargetNodes();
      foreach (d ∈ dsts) {
         bool dIsReady = true;

         /* Check if d is ready */
         List nested = same = exit = {};
         Set dsrcs = d.SourceNodes();
         foreach (x ∈ dsrcs) {
            if (IsNotBackEdge(x, d) && (x.visitID == -1)) {
               dIsReady = false;
               break;
            }
         }

         /* Add d to appropriate ready list */
         if (dIsReady) {
            if (d.loopDepth >  b.loopDepth) nested.Append(d);
            if (d.loopDepth == b.loopDepth) same.Append(d);
            if (d.loopDepth <  b.loopDepth) exit.Append(d);
         }
      }

         /* Nodes are visited in depth-first visit order. Therefore, nodes are inserted
         to the head of the ready list. In addition, amongst all ready nodes, nested
         nodes are visited first, and nodes outside the loop are visited last. */
      readyList.InsertListAtHead(exit);
      readyList.InsertListAtHead(same);
      readyList.InsertListAtHead(nested);
   } while (readyList.IsNotEmpty());
}
```

Figure 6.7. Algorithm to compute a basic block visit order suitable for loop specialization

However, with a small modification to the topological sorting algorithm, we can handle this scenario. Whenever there is a choice of multiple nodes to visit, the topological sort algorithm chooses the node which is nested deeper.

The algorithm for computing visit orders is shown in Figure 6.7. The computed visit order is a valid topological sort order on the CFG with back edges removed. However, the topological sort order satisfies the constraint that all nodes within a loop are visited before nodes outside the loop are visited. An additional property satisfied by the visit order is that the nodes of the loop will have a tight numbering (i.e. if the loop has n nodes, they will be numbered $x, x+1, ..., x+(n-1)$).

With the visit order computed by the algorithm shown in Figure 6.7, the SCCP algorithm will always process an entire loop before it processes the exit nodes of the loop.

Imposing a visit order on SCCP: Having shown a suitable visit order, we now show how to impose this visit order on `SCCP_Specialize_Basic`.

SCCP uses two work lists in its processing: a flow work list holds CFG (control-flow) edges for processing, and a ssa work list holds SSA (data-flow) edges for processing. Whenever a control-flow or data-flow edge is processed, its successors are added to the appropriate work list. While the SCCP algorithm does not specify any particular order for visiting and processing these edges, we now have to process edges according to the basic block visit order computed by the algorithm shown in Figure 6.7.

For a basic block `b`, `b.visitID` specifies `b`'s position in the visit order. These ids are used to assign ids to edges as follows. For a control-flow edge `e:x → y`, `ẽ.visitID = y.visitID`. For a data-flow edge `e:(p → q)`, (`e.visitID = p.basicBlock.visitID`).

Given two basic blocks `u` and `v` such that `u.visitID < v.visitID`, the visit order specifies that edges associated with `u` have to be processed before those associated with `v`. This translates to an equivalent rule for edges. Given edges `e` and `f`, such that `e.visitID < f.visitID`, `e` has

to be processed before f. Since SCCP always processes edge targets, and we assign edges the ids of their targets, this ensures that the basic block visit order is satisfied.

```
class EdgeSet {
   int visitID;  /* visit id for the edges in this set */
   Set edges;    /* e.visitID == visitID for all e ∈ edges; */
}

MemoryLocation Set SCCP_Specialize_Basic(CFG c)
{
   iml = {}; /* Set of specialized memory locations */
   while (edgeSetList is not empty) {
      EdgeSet es = edgeSetList.GetHead();
      while (es.edges ≠ φ) {
         Edge e = es.RemoveEdge();
         if (e is a CFG edge) ProcessCFGEdge(e, iml);
         else ProcessSSAEdge(e, iml);  /* e is a SSA edge */
      }
      edgeSetList.Remove(es);
   }
   return iml;
}

void AddEdge(Edge e:<u, v>)
{
   int id = (e is a CFG Edge) ? v.visitID : v.basicBlock.visitID;
   EdgeSet es = null;
   if (edgeSetList.EdgeSetExists(id)) {
      es = edgeSetList.GetEdgeSet(id);
   } else {
      es = new EdgeSet(id);
      edgeSetList.InsertInVisitOrder(es, id);
   }
   es.AddEdge(e);
}
```

Figure 6.8. Enforcing a visit order on SCCP_Specialize_Basic

Figure 6.8 shows the modified SCCP_Specialize_Basic algorithm which processes edges in the visit order. In order to do this, the algorithm collects all control-flow and ssa data-flow edges that have the same visit id in an edge set. As shown in the figure, all the edges within an edge set are

processed before a new edge set is processed.

Since the algorithm operates on edge sets, there is no need to maintain two work lists that SCCP uses (one for CFG edges and another for SSA edges). A single work list of edge sets suffices; the algorithm uses `edgeSetList` to record these edge sets. However, unlike SCCP, where the order of selecting elements from the work lists does not matter, `edgeSetList` is sorted by the visit ids of the member edge sets. Whenever new data-flow and control-flow edges are added to the work list (by `ProcessCFGEdge` and `ProcessSSAEdge`), they are either added to an existing edge set in `edgeSetList`, or added to a new edge set which is inserted into `edgeSetList` in sorted order. The `AddEdge` routine shown in Figure 6.8 maintains the sorted order of `edgeSetList`.

Unrolling loops and specializing them: Having presented several modifications to SCCP and the basic specialization algorithm, we now present an extension to `SCCP_Specialize_Basic` to specialize loops. This algorithm is called `SCCP_Specialize_Loops` and is shown in Figure 6.9. This algorithm is similar to `SCCP_Specialize_Basic` except for the addition of *line a* and *lines 0–8* marked in Figure 6.9.

Recall that this routine is invoked by the main specialization routine (`SCCP_Specialize` in Figure 6.1). Also recall from Section 6.2.2 that loops are specialized by peeling iterations one at a time and specializing the peeled iterations. We now discuss the extensions shown in Figure 6.1 that implement this loop specialization.

As shown in line 1 of Figure 6.9, loop iterations are only peeled when loop entry edges are processed (whose targets are loop headers). Assuming that the `ShouldPeelLoopIteration` routine in line 1 decides that a loop iteration can be peeled, the code in lines 2–8 is straightforward. The `PeelLoopIteration` routine creates a copy of the loop iteration, and updates the SSA property by renaming all variables defined within the loop. The algorithm then recursively invokes the specialization routine (line 3) by passing `s`, the current SCCP state of the algorithm. The SCCP

```
MemoryLocation Set SCCP_Specialize_Loops(CFG c, SCCP_State s)
{
   iml = {}; /* Set of specialized memory locations */
   while (edgeSetList is not empty) {
      EdgeSet es = edgeSetList.GetHead();
a.    if (CanUnrollNestedLoops(es.bb)) continue;
      while (es.edges ≠ φ) {
         Edge e = es.RemoveEdge();
         if (e is a CFG edge) {
0.          BasicBlock t = e.dst;
1.          if (e.isLoopEntry && s.ShouldPeelLoopIteration(e, t)) {
2.             CFG lc = PeelLoopIteration(t.innerLoop);
3.             SCCP_Specialize_Loops(lc, s);
4.             if (lc.LeadsToInfiniteUnrolling())
5.                s.IgnoreLoop(t.innerLoop); continue;
6.             else
7.                Splice lc into c;
8.          }
            ProcessCFGEdge(e, iml);  /* same as SCCP_Specialize_Basic */
         } else {
            ProcessSSAEdge(e, iml);  /* same as SCCP_Specialize_Basic */
         }
      }
      edgeSetList.Remove(es);
   }
   return iml;
}
```

Figure 6.9. Algorithm SCCP_Specialize_Loops

state s contains the mapping of variable names with their SCCP lattice values. Therefore, by passing s to the recursive invocation, the unrolled iteration is able to access the values of variables that are constant on entry to the loop. In addition, when the recursive invocation completes, the calling invocation is able to access the new constants that are discovered by specializing the peeled iteration. This is possible because the SCCP state s is shared among the recursive invocations.

After the peeled iteration is specialized, the algorithm uses the LeadsToInfiniteUnroll-

`ing` routine to check if any loop exit condition was specialized to a constant.[4]

If none of the loop exit conditions specialized to a constant value, the algorithm assumes that the loop unrolling process will not terminate and stops the specialization of the loop. Therefore, the specialized iteration is thrown away and the loop is ignored (future calls to `ShouldPeelLoop-Iteration` will return false for this loop). Note that since all variables in the specialized loop iteration were renamed prior to specialization, no fixup of the SCCP state `s` is necessary.

If at least one loop exit condition is a constant, the algorithm *heuristically* assumes that the unrolling process will terminate. In reality, it is necessary to use some cut-off metric to prevent infinite loop unrolling. Such a cut-off metric will be useful even when the loop unrolling process might terminate. If the `LeadsToInfiniteUnrolling` routine determines that the loop unrolling process will terminate, the specialized loop iteration is spliced into the CFG. If this happens to be the last iteration of the loop that will be executed, the residual loop body will become dead code and will be removed by a dead code elimination pass. If not, the loop header will be visited again which leads to further peeling and specialization.

During splicing, besides adding the control-flow subgraph of the specialized iteration into the CFG, variable renaming is also performed; variables used in μ-nodes[5] as well as variables defined within the loop and used outside the loop are renamed.

We now discuss how to decide if a loop iteration can be peeled (routine `ShouldPeelLoop-Iteration` in Figure 6.9). Whenever `SCCP_Specialize_Loops` processes a loop header, the `ShouldPeelLoopIteration` routine decides whether the loop should be unrolled and specialized using the FSM shown in Figure 6.10.

The state of the FSM is tracked by `s` (the SCCP-state variable in Figure 6.9). The state machine can be in one of four states: (default) *peel-loop* state, *specialize* state, *no-peeling* state, *delay-peeling*

[4]Note that a loop can be exited from multiple places within a loop. There will be one loop exit condition corresponding to every `break` statement.

[5]ϕ-nodes in loop headers

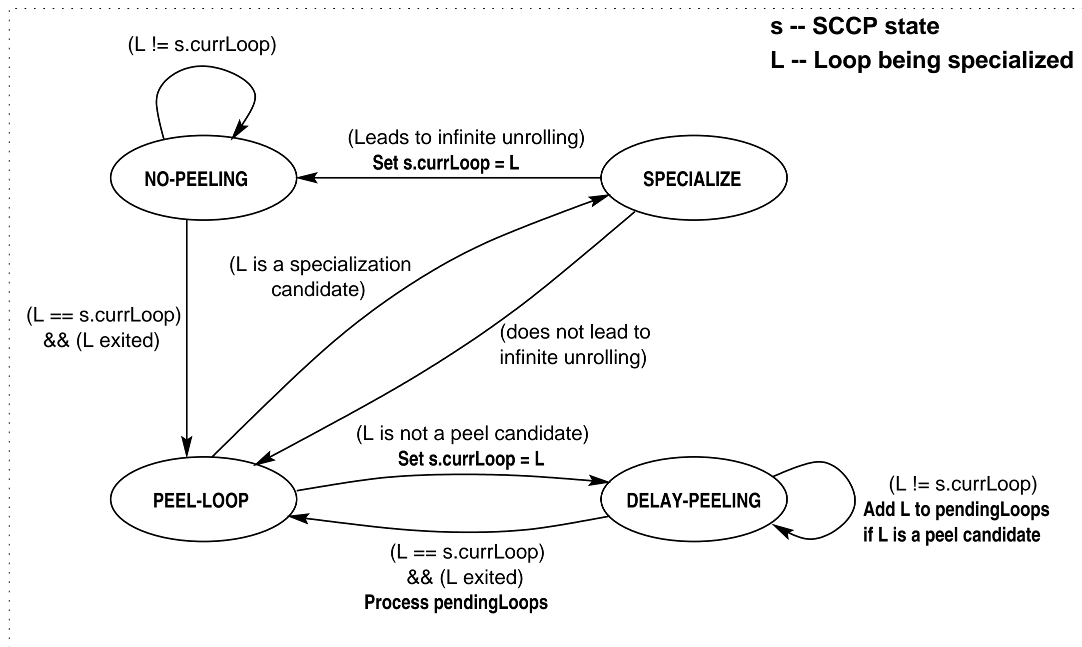

Figure 6.10. State machine used to unroll and specialize loops

state. The FSM allows loop peeling and specialization only if it is in the *peel-loop* state and the loop is a specialization candidate (decided by the scope-building algorithm). If so, the FSM transitions to the *specialize* state. In this state, the loop is peeled and specialized (lines 3 and 4 of Figure 6.9). Based on the check on line 4, the FSM now transitions to either the *peel-loop* state or the *no-peeling* state (line 5). In the *no-peeling* state, all nested loops are ignored even if the spec-scope algorithm has identified them for unrolling. This is based on the (untested) heuristic that peeling of a nested loop is beneficial only when the specialized context of the outer loop is available. But, since the specialization of the outer loop failed, it is assumed that specializing the inner loop is not useful. The FSM enters the *peel-loop* state from the *no-peeling* state on exit from the loop for which specialization failed.

Next, let us consider the *delay-peeling* state. The FSM enters this state when `SCCP_Specialize_Loops` encounters a loop which has not been marked for unrolling. In this state, unrolling of

all nested loops is delayed till the loop is exited. Since the algorithm processes all edges contained in the loop before processing any outside edges, it processes an external edge only after the SCCP state for the loop reaches a fixed point. Therefore, once `SCCP_Specialize_Loops` exits the loop, it is guaranteed that the SCCP state for the loop will be the correct state. Once this loop is exited, pending loops (nested loops that are also specialization candidates) are considered for specialization. This is shown on the transition from the *delay-peeling* to *peel-loop* state. This is also shown as the check in *line a* in Figure 6.9. Processing pending loops involves adding the loop entry edges for these loops into the `edgeSetList` worklist. When these edges are picked up, the nested loops will be unrolled (if marked for unrolling).

This completes the description of our specialization algorithm that integrates loop unrolling and specialization with SCCP. We now present a discussion of the time complexity of this algorithm.

Time complexity of the algorithm: Let us first consider the complexity for acyclic CFGs. Topological sort order has a linear time complexity $O(|V|+|E|)$ as does the main SCCP routine. However, `SCCP_Specialize_Loops` can potentially have a worst-case complexity of $O(|V|^2)$ whereas SCCP has a worst-case complexity of $O(|V| + |E|)$. The $|V|^2$ term arises because the algorithm inserts edge lists in sorted order in the `AddEdge` routine in Figure 6.8. But, this worst-case complexity be avoided by ignoring the visit order for acyclic CFGs.

For cyclic CFGs, the worst-case upper bound for the algorithm is $O(|V|^2 + (m|V|)^{2^{d+1}-2})$ where m is the highest unrolling count of any unrolled loop and d is maximum loop nesting depth that is unrolled and specialized. For example, if only one loop within a loop nest is unrolled and specialized, the value of d will be 1, and the worst-case upper bound will be $O(|V|^2 + (m|V|)^2)$.

Thus, this loop specialization algorithm has an exponential unrolling cost. This is because the algorithm processes loops from outside to inside one iteration at a time. Therefore, if the outer loop has m iterations, and the inner loop has n iterations, there will be $m * n$ total unrollings. In contrast,

if the algorithm could process loops inside to outside, the exponential cost will disappear. However, unrolling loops from inside to outside is not always possible because the loop count of the inner loop might depend on the specialization of the outer loop.

This algorithm could potentially be improved to minimize unrolling overheads for nested loops. However, for our evaluation, the maximum value of d for two, and it was one for most benchmarks. Chapter 9 presents an empirical evaluation of the specialization cost.

6.2.3. Comments on the Loop Specialization algorithm

Having presented the loop specialization algorithm, we make some observations about it.

- While it was not necessary in the experiments we carried out, it might be useful to use some cut-off metric to prevent code bloat due to unrolling.

- The loop specialization algorithm can also be implemented using a structural dataflow analysis algorithm like interval analysis or T1-T2 algorithm [6]. Using such algorithms, CFG transformations like loop unrolling can be performed before the analysis terminates using the intermediate state of the analysis.

- In our algorithm, specialization of a loop iteration fails *only when* it might lead to infinite unrolling, i.e., when the loop exit condition is not a specialization-time constant. However, one might decide to abort unrolling earlier using a cost-benefit analysis. If the cost-benefit analysis of the specialized iteration reveals that unrolling was not beneficial, the loop specialization process can be aborted.

- In our technique, only loops selected by the spec-scope finding algorithm (Figure 5.1) are specialized. However, another option could be to attempt specialization of *every* loop in a scope and use a cost-benefit analysis to decide whether to continue unrolling or abort unrolling.

While this technique might potentially result in better specialization, the drawback is that this can lead to higher specialization overheads.

- Another completely different approach to loop unrolling would be to use profiling to determine the unrolling factors for different loops. These profiles can be used to unroll the loops prior to performing SCCP. However, this approach will work well only when the loop unrolling count is independent of the lookup key. Otherwise, it will be hard to determine the unrolling count for different lookup keys. This is a problem, for example, when unrolling for recursive data structures like linked lists.

6.3. Related Work

Muth *et al* [73] have previously attempted using constant propagation for performing compile-time specialization of programs based on value profiles. They perform a similar CFG transformation as shown in Figure 3.2. However, their lookup is organized as a binary search tree coded using `if-then-else` chains whereas we use several low-overhead lookup techniques which are discussed further in Chapter 7. Another significant difference between the two techniques is in the implementation of loop unrolling. The algorithm presented in this chapter integrates loop unrolling with SCCP which enables it to perform much better specialization. In the work by Muth *et al.*, unrolling is determined by profiling loop index expressions. As such, this technique will only be able to unroll loops that have simple loop indexing.

In contrast with existing specializers based on the offline partial evaluation technique [8, 30, 48, 75], our specializer is a hybrid technique. By using the scope-building algorithm to identify scopes and loop unrolling candidates, our specializer resembles offline partial evaluators. However, the specialization itself that is carried out within a spec-scope resembles an online partial evaluation strategy.

However, the loop unrolling technique implemented in this dissertation is weaker when compared to DyC [47]. DyC implements multi-way unrolling, whereas we do not implement this. For simple loops, such as those that merely increment a counter or those loops that traverse linear linked lists, the unrolling technique described in this chapter can completely unroll and specialize them. However, for other complex loops where, there are multiple alternatives for the next iteration, our specialization technique cannot unroll the loop. An example of such loops are interpreter loops where the next pc depends on input data and can be one of multiple values. DyC specializes these kind of loops by creating multiple alternative specialized loop iterations. In general, DyC generates a directed graph of specialized loop iterations. This is called multi-way unrolling.

CHAPTER 7

IMPLEMENTING LOOKUPS: DISPATCHING TO SPECIALIZED CODE

In the specialization model presented in Chapter 3, given a scope $SS(k)$, the specializer creates multiple specialized versions of SS, one for each value of the key k. A `lookup` instruction uses the value of k to *dispatch* (transfer control) to the appropriate specialized version.

At a conceptual level, the dispatch can be implemented by searching a table of specialized versions using the value of k as the search key. At a detailed level, the lookup table can be implemented in many ways: as a single centralized software table, as a single centralized hardware table, or as a distributed search table (which is not a table at all). In this chapter, we discuss some implementations of the dispatching mechanism.

7.1. Centralized software hashtable

In this straightforward implementation, a centralized software hashtable is used to implement the lookup table. The dispatch is then implemented as a hashtable lookup that uses `<k, p>` as the hash-key, where `p` is the pc of the lookup instruction. These lookups can be expensive because of the cost of (i) generating the hash-key (ii) indexing the hash table using the hash-key, and (iii) traversing collision chains. This dispatching mechanism is presented only for the sake of completeness and our specializer does not use this technique. Instead, it implements other more inexpensive dispatching mechanisms which we discuss in the following sections.

7.2. `if-then-else` chain

This technique implements the lookup instruction as a chain of `if-then-else` statements. For example, if there are two specialized versions `S1` and `S2`, corresponding to `k` = o_1 and `k` = o_2, the dispatch is implemented as:

```
if (k == o1) goto S1;
elseif (k == o2) goto S2;
else goto default-version;
```

Referring back to our conceptual model of the lookup table, note that this implementation can be considered to be an inlined implementation of a distributed lookup table. This technique is similar to the implementation of Polymorphic Inline Caches (PICs) in Self and other object-oriented languages to minimize virtual call overheads [54].

To minimize dispatching overheads, the checks are ordered in descending order of the expected execution frequencies of the specialized versions. The specializer consults the object-access profile and uses the profiled frequencies of the specialization keys as an estimate of the expected execution frequencies of the corresponding specialized versions. In the example above, if the object-access profile indicates that o_1 is seen 30% of the time, and o_2 is seen 50% of the time, the specializer will reverse the order of the checks above.

The cost of dispatching to a specialized version accumulates as the search proceeds down the chain. Thus, the dispatching cost is the lowest for the most frequently executed specialized version and is the highest for the default case. Therefore, this technique is best implemented when there are a *small* number of specialized versions.

An alternative to implementing the dispatch as a linear chain would be to implement this as a binary search tree. However, our specializer does not find a need for a binary search tree since the `if-then-else` implementation is considered only when the number of specialized versions is less than three. When the number of specialized versions if greater than three, dispatching mechanisms presented in the following sections are used.

In contrast to the two lookup implementations we presented which have variable dispatching costs, the two implementations we present next have a constant dispatching cost.

7.3. Scopes as pseudo-methods

In this implementation, the specializer tries to dispatch to the specialized version of a scope using the virtual call mechanism (where it applies).

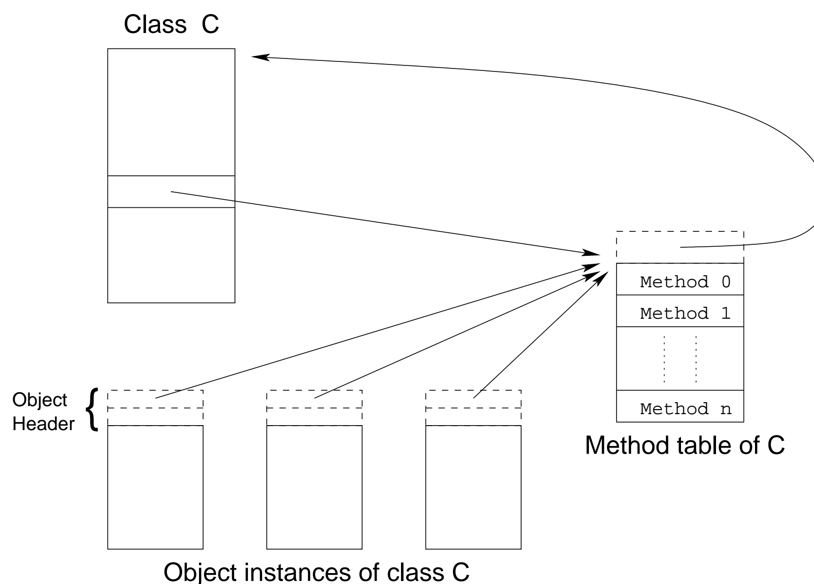

Figure 7.1. Object implementation assumed by the dispatch implementations specific to Java (and potentially other OO-languages)

This implementation is specific to Java (and potentially other object-oriented languages) and assumes an object implementation in which object headers have a pointer to the shared method table of the defining class. This object implementation is shown in Figure 7.1. As shown in the figure, the header of objects of class `C` point to a method table that is shared between them. The method table has a hidden pointer to the JVM-specific internal class structure.[1] Thus, method tables are one object reference away and the defining class is two object references away.

To do this, the specializer considers a scope `SS(O)` to be a pseudo-method of `O`'s class and considers the specialized versions of `SS` to be the inherited methods of the default version of `SS`. Let $SS_1 ... SS_k$ be the specialized versions corresponding to key values $o_1, ..., o_k$. Suppose these

[1]This class structure is also a regular Java object.

objects are instances of class `C`. The specializer promotes `SS` to the status of a method of class `C` by allocating it a method table entry in `C`'s shared method table. Referring back to the figure, the specializer allocates a new slot at position $n + 1$ in the shared method table. This entry is filled with the address of the default (unspecialized) version of `SS`.

To implement the dispatch, the specializer creates a clone of the shared method table for each o_i and modifies the object header of o_i to use the cloned method table. The $(n + 1)^{th}$ slot in this cloned method table (of o_i) is modified to point to SS_i. This modification is shown pictorially in

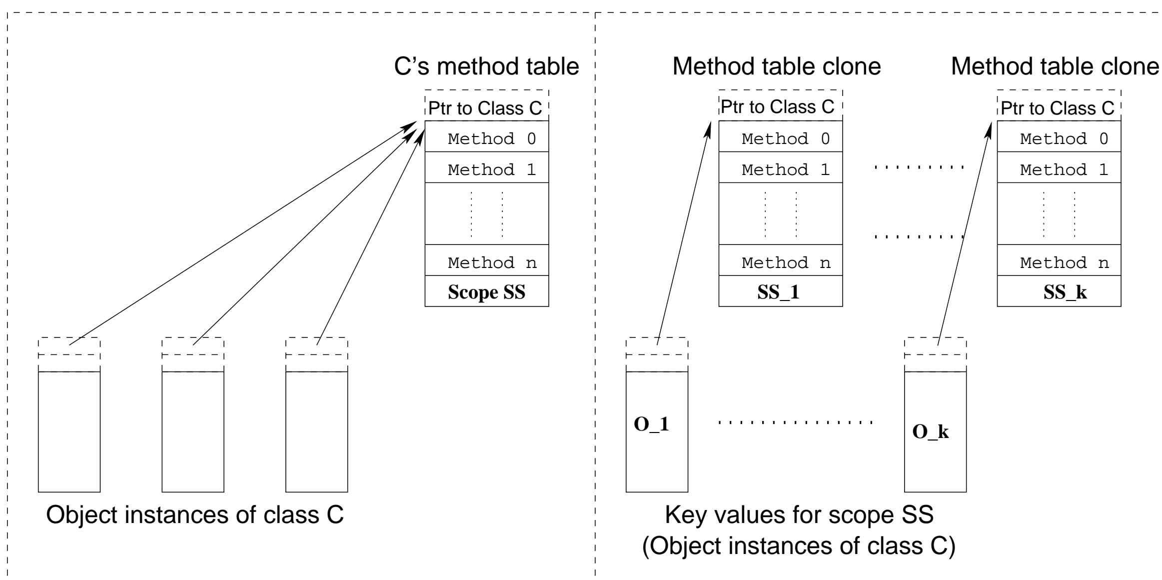

Figure 7.2. Implementing a specialization scope as a pseudo-method

Figure 7.2. With this modification, the lookup instruction `lookup(O)` is implemented as `jump O.mtbl[n+1]` which corresponds to the 3-instruction sequence shown below:

```
r = load O.mtbl;
a = load r[n+1];
jmp a;
```

This dispatch implementation has a fixed cost for all specialized versions and is suitable even when there are a large number of specialized versions. However, there is a space cost due to the

duplication of method tables. Greater the number of specialized versions, greater the space cost. In this sense, this dispatch implementation trades off space cost to reduce time costs. Depending on the footprint of the duplicated method tables, the extra space might manifest as increased execution time due to increased data cache miss rates.

Note that the pseudo-method technique works for array keys too, at least in Java. In Java, all array objects are instances of `java.lang.Object` and inherit the methods of this class. In addition, this implementation works only when the specialization key is a single-element object reference. Since our specialization model makes this exact restriction, this lookup implementation technique can be used in all cases.

7.4. Piggybacking onto the method virtual table

This is an optimization of the pseudo-method implementation when: (1) `O` is the receiver of method `m`, and (2) `SS(O)` is the only scope in method `m`.[2] In this scenario, the specializer expands the scope to the entire method and creates specialized methods $m_1, ..., m_k$. As in the previous implementation, method table clones are created for each specialized version. However, instead of creating a new pseudo-method slot for `SS`, the existing method table entry for `m` is used (since `SS = m`). For the key value o_i, the method table entry for `m` in o_i's cloned method table is changed to point to m_i. With this change, when `m` is invoked as `call O.m()`, o_i transfers control to the specialized method m_i. Unlike the other implementations, no extra instructions are required to implement the lookup. However, the drawback is that there can be unnecessary code duplication – instructions of the method not in `SS` are also duplicated.

However, this implementation is not always suitable since this relies on `m` being invoked with

[2]This condition is strictly not necessary for correctness. However, this check is used to prevent excessive code duplication. Consider the case where `m` has two scopes: S_1 with n_1 versions and S_2 with n_2 versions. Assuming that the key of S_1 is the method receiver, n_1 versions of `m` are created. In each of these n_1 versions, n_2 versions of S_2 would have to be created. Thus, this leads to $n_1 * n_2$ copies of S_2 rather than just n_2 copies.

an indirect call rather than a direct call. If the dynamic optimizer optimizes an indirect virtual call with a direct call (based on type analysis), then control will not be transferred to the specialized version. Therefore, this implementation should be considered an optimization of the pseudo-method technique and requires the optimizer to keep track of direct calls.

7.4.1. Chaining of specialized versions

An additional advantage of this dispatch implementation is that in certain cases, dispatching via virtual call can be eliminated with a direct call – this has the effect of chaining together specialized versions.

Consider the `FindTreeNode` code shown in Figure 1.1. In a specialized version of the method, each iteration of the unrolled loop will (recursively) invoke the `FindTreeNode` method. Some of these calls might correspond to specialized versions of the method. For the calls for which specialized versions exist, the virtual call dispatch can be replaced with a direct call to the specialized version of the method.

7.5. Choosing a lookup implementation

Since the specializer has a choice of lookup implementation techniques available, it needs a decision procedure to pick the appropriate one. As must be clear from the preceding discussion of the different lookup implementations, the pseudo-method implementation is the preferred technique because it is applicable in all cases (with the specialization model studied in this dissertation) and has a fixed lookup cost independent of the number of specialized versions.

In our evaluation, the `if-then-else` chain is used only when the number of specialized versions is less than three. In all other cases, the pseudo-method implementation is used. In those cases where the method table piggybacking optimization can be performed, the specializer uses it.

7.6. Related Work

Dispatching techniques have been studied previously in other contexts. For example, most compilers for the C programming language implement some switch statements (where the case labels are densely packed) using jump tables. The pseudo-method implementation is, in some ways, analogous to this technique, but, is different in other ways. Firstly, jump-table implementations include a range check to jump to the default case. Secondly, the jump-table implementation is most useful when the case labels are not sparsely distributed (Ex: 0, 10, 3045, 10034). In this example, a jump-table implementation would be very expensive in terms of space since a table with 10035 entries would have to be allocated. In contrast, a pseudo-method implementation is not impacted by this constraint since it piggybacks onto an already-existing virtual method table implementation of virtual method calls.

The pseudo-method dispatch implementation is similar in spirit to the to the distributed memo-table implemented in data specialization [58] where new fields are added to objects to store the result of commonly executed invariant expressions. However, while data specialization adds data fields to objects to record values of expressions, the pseudo-method technique studied in this dissertation adds method address fields to method tables to record code addresses. These two techniques are orthogonal and can be used in the same implementation of a specializer.

7.7. Future Work

When the restrictions on the specialization model are relaxed as part of future work, other lookup implementations will have to be considered since the pseudo-method technique will not always be applicable. This is left for future work when more powerful specialization models are studied.

Also left for future work is potential hardware implementations of lookup tables. Hardware reuse techniques have been previously proposed that rely on memo table support for exploiting com-

putation reuse. For example, Sodani and Sohi [84] study instruction reuse techniques which relies on a centralized hardware reuse table in the processor core. Connors and Hwu [29] study more coarse-grained compiler-directed computation reuse techniques. They rely on sophisticated hardware support for efficient dispatch to reusable computation regions. While special-purpose hardware for specialization is unlikely, general-purpose memo table support might be potentially useful. For this support to be low-overhead, the hardware would have to support index computation via hashing of the lookup keys. Such memo-table support might be used by optimizers (static or dynamic) to reuse computations at varying granularities. For example, it could be used by specializers (like that studied in this dissertation) to lookup addresses of specialized versions. Or, it could be used by regular memoization optimizations typically implemented in software. The exploration of this design space is left for future work.

CHAPTER 8

ENSURING CORRECTNESS OF SPECIALIZED CODE

Section 3.5.1 presented the store-profile based profiling analysis used by the specializer to identify invariant parts of runtime data structures. However, since a runtime profile only captures past program behavior, the specialized memory locations might be written at a later point in the program's execution. Such writes may invalidate[1] all specialized code that rely on the invariance of the written memory locations. Therefore, the specializer has to provide mechanisms to guarantee the correctness of specialized code in the face of such writes to specialized memory locations.

This problem is not unique to our specialization system. Any specializer that attempts to exploit invariance in runtime data structures faces this problem. For example, Tempo [31] relies on full-program alias analysis to detect specialization opportunities in the presence of pointer manipulation and side-effects. Tempo uses this analysis to identify potential invalidation locations. In cases where Tempo does not have access to source code, like libraries, it relies on the programmer to specify the alias relationships and side-effects of that code. As another example, DyC [47] relies on the programmer to annotate invariant memory accesses as well as annotate program points that can invalidate specializations that depend on them. While Calpa [70] generates these annotations for DyC automatically, as we noted in Chapter 2, the overhead of Calpa's analysis prevents its efficient deployment at runtime.

In the context of a dynamic optimization system, the requirements of low overheads and transparency render the above approaches infeasible. Given the requirements of transparency, a specializer cannot rely on programmer annotations to guarantee correctness. Neither can it rely on full-program alias analysis because low-overhead scalable versions of this analysis are not available to be implemented at runtime. However, variants of the alias analysis approach can be implemented

[1]Note that if the store does not change the contents of the memory location, there is no need to invalidate. Stores that do not change the content of the memory locations being written are known as *silent stores* [62] and have been used to reduce coherence traffic in multiprocessor applications.

for Java to detect invalidation of specialized code.

This chapter proposes mechanisms for guaranteeing correctness of specialized code. Before presenting techniques that can detect potential invalidations (due to modification of specialized memory locations), we first describe the invalidation process since this process is independent of the specific detection technique. We then present several solutions to detect modifications of specialized memory locations.

8.1. Implementing Invalidation of Specialized Code

Since invalidations are expected to be rare, the cost of invalidation is insignificant. We now present a simple technique for implementing invalidations.

Whenever the specializer detects a store to a specialized memory location, only a subset of the specialized versions need to be invalidated. To do selective invalidation, the specializer maintains a mapping from specialized memory locations to dependent specialized versions. This mapping can be maintained using a binary search tree or a hash-table.

Suppose that the specializer detects a store to a memory location `L` with address `A`. The address `A` is used to fetch the specialization scope `S(k)` and the specific specialized version `SV` that uses `L`. The details of invalidating `SV` depends on the specific lookup mechanism implemented by `S(k)`. If `S` used the method table of `k` (either by creating pseudo-methods or by using actual methods) to implement the lookup, the specializer can invalidate `SV` be reverting the method table entry of `O` (`SV`'s key value) to point to the default version of `S`.[2] If `S` used a `if-then-else` chain to implement the lookup, then the specializer can invalidate `SV` by changing the appropriate jump address to point to the default version of `S`. If `S` relied on a centralized software hashtable, then `SV` can be invalidated by removing `SV` from the hashtable.

[2]Note that the method table pointer of `O` could also be reverted to point to the default method table. While this is a solution maintains program correctness, this is not a desirable solution because other entries in `O`'s cloned method table could be pointing to other specialized scopes.

Understanding how to implement invalidations, let us now see how to detect invalidations in the first place.

8.2. Detecting Invalidation of Specialized Code

In the context of a dynamic optimization system, coupled with the invalidation mechanism presented in the previous section, a specializer can guarantee correctness of specialized code using one or more of the following approaches:

- *Guaranteeing invariance of specialized memory locations*: In this approach (*invariance guarantees*), software analysis guarantee that the specialized memory locations will not be written after they are specialized. The advantage of this approach is the low overheads: the cost is paid only once – during specialization. The disadvantage is that this approach is not widely applicable and other approaches would have to be used to guarantee correctness of other specialized memory locations. In this chapter, we present a solution that exploits the semantics of Java to guarantee the invariance of a restricted set of specialized memory locations.

- *Guarding specialized memory locations*: In this approach (*memory guarding*), the specializer guards the specialized memory locations to detect writes to these locations. This guarding can be accomplished with custom hardware support (ex: protection bits in memory – like error-correction bits), or with OS-based support. The advantage of this approach is that it is widely applicable. The disadvantage of this approach is that there is no existing low-overhead memory guarding support available. In this chapter, we present a memory guarding solution based on a recently proposed OS-based fine-grained memory protection schemes – but, this is still a research proposal is not available in existing systems.

- *Monitoring stores to detect writes to specialized memory locations*: In this approach (*store monitoring*), all stores that could potentially write into specialized memory locations are mon-

itored. In Java, software solutions for store monitoring involves instrumenting a subset of all `putfield, putarray` bytecodes in the program that could potentially write into specialized memory locations. The advantage of this approach is that it is widely applicable and can be implemented in existing Java Virtual Machines. The disadvantage of this approach is that it introduces overheads in the execution of non-specialized code. In this chapter, we present a software solution for store monitoring and present a qualitative discussion of its overheads. One could potentially think of hardware-based approaches for store monitoring. However, hardware-based approaches have not been studied in this dissertation.

- *Verifying specialized memory locations at scope entry*: In this approach, rather than monitor stores or guarding specialized memory locations, the content of specialized memory locations is verified at the entry of a specialized version. The advantage of this approach is that non-specialized code does not incur extra overheads and the cost of detecting invalidations is easily quantifiable and can be used in the cost-benefit model. Furthermore, it can be implemented in existing JVMs. However, this approach is only applicable in restricted scenarios – when a specialized version only uses a much smaller set of specialized memory locations when compared to the expected benefit from the specialized version. This approach has not been studied in this dissertation and is left for future work.

Having presented several approaches to detecting modifications to specialized memory locations, we present techniques that follow the above approaches. We first present a memory guarding technique.

8.3. Memory Guarding via fine-grained memory protection support

Recently, Witchel *et al.* [90] proposed Mondrian Memory Protection, a fine-grained memory protection scheme implemented in the operating system that relies on hardware support. In this

scheme, permissions are granted to memory segments which can be as small as individual words. Using this scheme, the specializer can grant read-only permissions to specialized memory locations (the set `iml` in Figure 3.4). Whenever these memory locations are written, the memory protection scheme traps to software which invalidates the specialized versions that used these locations.

Thus, the Mondrian scheme is perfectly suited for our purposes and does not require any software analysis or instrumentation of non-specialized methods. While this scheme is not yet available in current systems, fine-grained memory protection has many uses [90] and future OS and hardware systems might provide such support. For example, Transmeta Crusoe [57] processors have custom hardware support to enable aggressive load speculation performed by their code morphing software. To detect failure of these speculations, these processors have hardware that detects writes to memory locations that were optimized away.

8.3.1. Overheads: Qualitative discussion

In order to study the space and time overheads of the Mondrian Memory Protection scheme, the authors experimented with two extreme cases. In the first experiment, they tried to compare their scheme with existing virtual memory operating systems with coarse-grained memory protection schemes. For this experiment, the number of memory segments were under 50. In this experiment, the space overheads were under 0.7% and the number of extra memory references introduced were lesser than existing virtual memory schemes. Thus, their scheme can efficiently support coarse-grained memory protection.

At the other extreme, where they attempted to stress the fine-grained memory protection scheme, new memory segments were created for every object allocated by the application. For allocation-intensive applications (Java benchmarks), over 1.5 million segments were created. For this experiment, the space overheads were under 9% and the number of extra memory references were under 8%.

In contrast with this extreme case, our specializer had to guard at most 8KB memory for the programs we used to evaluate our specialization system (see Table 9.1). In this extreme case, the 8KB specialized memory was spread over 830 objects and 500 arrays. Compared to the 1.5 million segments used to stress MMP, this is a small fraction. Therefore, we believe that memory guarding using fine-grained memory protection is an attractive solution where such support is available.

8.4. Software Store Monitoring Schemes

We now present a software-based invalidation-detection technique that is based on the store-monitoring approach. This software store monitoring technique can be implemented in existing Java Virtual Machines to monitor stores to detect writes to specialized memory locations. This technique relies on Java semantics to do this efficiently. The central idea is to instrument putfields and putarrays that could write into the specialized memory locations. We first present a baseline store monitoring technique and then present various optimizations which help reduce the overheads store monitoring. One of these optimizations is based on the invariance guarantee approach.

8.4.1. Baseline store monitoring

We first present a store monitoring technique for non-array objects and then extend it to handle arrays.

Let $SO = \{o_1, o_2, ...o_n\}$ be the set of objects that had at least one of their fields specialized. Let $F_s = \{f_1, ..., f_m\}$ be the specialized fields of objects in SO.[3] Pick an arbitrary field $f \in F_s$ and proceed. Let $O(f) \subset SO$ be the set of objects that assume the invariance of field f. Let $PF(f)$ be the set of putfield sites that modify f. Since Java does not have pointer arithmetic, $PF(f)$ can be precisely computed for all $f \in F_s$. An optimizer thread could compute this information during specialization. Alternatively, $PF(f)$ sets can be incrementally computed (for every f in the

[3] $F_s = \{f \mid \exists\, o_i \in SO$ such that $o_i.f$ was assumed invariant$\}$.

application) at class loading time and incrementally updated during optimizations (like inlining). So, given $f, O(f)$, and $PF(f)$, in order to ensure correctness, all putfield sites in $PF(f)$ have to be monitored to detect any modifications to objects in $O(f)$.

The specializer can implement these monitors cheaply using a two-step process described below. First, all objects in $O(f)$ are marked. Object marking can be implemented by utilizing an unused bit in object headers. Second, all putfield sites in $PF(f)$ are instrumented to detect stores to marked objects. This can be implemented with a 3-instruction snippet shown in Figure 8.1. The putfield

```
x = load obj.spec_mark;
y = x | SPEC_MASK;
if (x == y) invalidate(obj, f) /* obj.f is being overwritten */;
```

Figure 8.1. Store monitor inserted before the instruction: `putfield (value, obj.f)`

monitor checks if the object being written is marked, and if so, it invokes an invalidation routine which invalidates the relevant specialized code using the technique presented in Section 8.1.

For arrays, a similar analysis can be carried out. If SA is the set of specialized arrays, then, depending on the types of arrays in SA, putarray sets can be computed for each array type (similar to the putfield sets for each specialized field). As in the case of putfields, monitors are inserted at all sites in these putarray sets to detect stores into marked array objects. The store monitor is shown in Figure 8.2. However, the putarray monitor cannot distinguish between writes to individual array

```
x = load a.spec_mark;
y = x | SPEC_MASK;
if (x == y) check_and_invalidate(a, i) /* a[i] is being overwritten */;
```

Figure 8.2. Store monitor inserted before the instruction: `putarray (value, a[i])`

elements. If all the elements of the marked array were used in specialization, then no further checks are necessary and the relevant specialized code can be invalidated. However, if only a few individual elements of the marked array were specialized, blanket invalidation will lead to lost performance. Therefore, the specializer checks whether the array element being written was specialized. Clearly,

this is a more expensive store monitor than that for putfields. Another option would be to conservatively invalidate all specialized code that used any element of the marked array. While this would reduce overheads of the monitor, this can potentially lead to lost performance. This is similar to the problem of false hits in cache coherence schemes in multiprocessor systems.

Having looked at the baseline scheme, in the following sections, we present optimizations of this technique which helps reduce overheads.

8.4.2. Optimization 1: Guaranteeing invariance through program analysis

This optimization of store monitoring is based on the approach of guaranteeing invariance of specialized memory locations.

Referring back to F_s, the set of specialized fields, if $f \in F_s$ is a field with a `final` declaration or if it is a `private` field that is modified only in constructors or in methods accessible only from constructors, language semantics guarantees that f will not be modified after initialization. All such fields can be safely eliminated from F_s. Example of such fields are the fields of String objects (instances of the `java.lang.String` class) which are immutable. If all fields are eliminated from F_s, the specializer has a guarantee that no specialized location will be overwritten.

8.4.3. Optimization 2: Reducing the size of the putfield/putarray sets

The previous optimization shrinks F_s, the set of specialized fields. The same analysis can also be used to shrink the putfield sets of the fields in F_s. Given a field f, during the computation of its putfield set, $PF(f)$, putfield sites in constructors or in methods that are reachable only via constructors can be eliminated. These putfield sites correspond to object initialization and can be safely ignored.

Other analysis could also reduce the size of the putfield/putarray sets. Referring back to the description of the baseline store monitoring scheme, for a field f, the specializer has to monitor

putfield sites in $PF(f)$ to check if they write to objects in $O(f)$. Therefore, all putfield sites that write into newly allocated objects can be removed from $PF(f)$.

8.4.4. Optimization 3: Integrating store monitors with write barriers

The store monitors shown in Figure 8.1 and Figure 8.2 resemble write barriers used by some concurrent and generational garbage collectors. Write barriers perform a similar function for incremental garbage collectors (generational, concurrent, and other flavors) as store monitors do for our specializer. For example, generational garbage collectors place write barriers around *all* pointer-updating stores to detect inter-generational pointers.

As a more specific example, in Figure 8.3, we show a write barrier used by Heil [50] to detect heap modifications made by the application in parallel with the concurrent garbage collector. In

```
mtp     = load obj.mtp;     /* load method table pointer */
liveMtp = mtp | LIVE_MASK; /* check for live mark */
if (mtp ≠ liveMtp) RecordObject(obj);
```

Figure 8.3. Write barrier used in a concurrent garbage collector by Heil [50]

general, many write barrier implementations have a fast path consisting of a few instructions (between 2 and 3) and a slow path consisting of more elaborate checking. The fast path is typically inlined at the store site.

For this example, store monitors at pointer-updating putfield sites can be combined with the write barriers at those sites and is shown in Figure 8.4. This requires that the object marking bits for the store monitor and the write barrier be allocated in the same header word. This code loads the

```
w = load obj.marks;                     /* load object marks */
x = w & (SPEC_MASK | LIVE_MASK); /* check for specialize and live marks */
if (x ≠ LIVE_MASK) ProcessIntegratedMonitor(w, obj, f);
```

Figure 8.4. Integrated store monitor and write barrier

common header word and does a single check to test for both bits simultaneously. This integration

essentially eliminates the overhead of the store monitor (or the overhead of the write barrier – if one is looking at it from the viewpoint of a garbage collector). Similar integration might be possible for other write barrier implementations.

8.4.5. Overheads: Qualitative discussion

Due to infrastructural limitations, invalidation schemes have not been implemented in the specializer studied in this dissertation. Consequently, this dissertation has not performed an experimental evaluation of the store monitor schemes. However, in the following sections, we make several qualitative arguments to judge the utility and overheads of store monitors.

Assuming that writes into specialized memory locations are rare, the checks in the store monitors shown in Figure 8.1 and Figure 8.2 will fail with high probability. This common case will be quickly learnt by branch predictors on modern out-of-order superscalar processors. As a result, the check snippets will not be on the critical path of execution since the branch predictor will successfully predict the result of the check. The processor uses the actual execution to verify the results of the prediction. This reliance on non-criticality of the store monitor is similar to the technique used by Heil [50] to perform speculative object inlining (called object co-location). Once an object `x` is inlined in object `y`, the load of `x` from `y` can be performed using a simple add operation (`y + offset(x)`). Since the inlining is speculative, correctness is verified by checking code. As in our example, the branch is highly predictable.

In order to better understand the overheads of store monitoring, we turn to studies of write barrier schemes in concurrent and generational garbage collection. As we noted earlier, store monitors resemble write barriers. By examining the overheads of various write barrier implementations, we can get an idea of store-monitor overheads. For Smalltalk programs, Heolzle [52] reports a maximum of 8% overhead using a fast 2-instruction store barrier. For Lisp programs, Zorn [92] reports a maximum of 6% overhead for a write barrier. We interpret these results to be indirect evidence for

the low overheads of the store monitoring solution. Implementation of invalidation techniques and evaluation of store monitoring technique is left for future work.

Evaluation

For most of our benchmarks (Table 9.1), we found that the software analysis we discussed in Section 8.4.2 can guarantee the invariance of all specialized memory locations without any store monitors. However, for some of the benchmarks, memory guarding or store monitoring is required for correctness. As we discussed in the previous sections, these can be implemented with low overheads using hardware/OS fine-grained memory protection schemes or using software monitoring techniques.

Synchronization Issues

Before concluding this chapter, we discuss a synchronization issue that needs to be addressed. Consider the following race condition which may arise when our specialization requires placing a memory guard (or store monitor). Assume that the specializer runs concurrently with the application. First, we specialize a scope S, using the content of a memory location a as an implicit input to specialization. Next, the invalidation algorithm determines that we need to guard a against writes, and turns on such a guard. Now, if the application overwrote a between the specialization and the activation of the guard, the specialization may be incorrect. Our solution is to verify that after the guard is activated, the guarded locations contain the same values as assumed during the specialization.

CHAPTER 9

EXPERIMENTAL EVALUATION

In this chapter, we present an experimental evaluation of our specialization technique. In Section 9.1, we describe the research infrastructure used to implement the specializer, present benchmarks and describe the evaluation methodology used to evaluate the specializer. In Section 9.2–Section 9.4, we evaluate the key properties of our specialization technique. In Section 9.5, we present the final speedup results for the various benchmarks which includes an estimate of runtime specialization costs.

9.1. Research infrastructure and Methodology

This section describes the research infrastructure used to implement the specializer and the methodology used to conduct our experiments.

This dissertation uses Strata,[1] a static-compiler based JVM, for implementing the specializer and evaluating it. Strata does not support multi-threading and supports only a subset of the JDK1.1 API. Strata does not have any direct support for runtime compilation. However, it supports profile-driven recompilation which we use to emulate the process of runtime specialization. We describe this emulation process in Section 9.1.2.

Figure 9.1 shows a pictorial representation of the Strata VM which consists of the Strata compiler and the Strata runtime system. The Strata compiler translates Java bytecodes into Sparc V8 assembly code, which is linked together with the Strata runtime system and standard libraries to produce a stand-alone executable. The Strata runtime system contains object allocation routines, garbage collection routines, I/O primitives, the Java Native Interface (JNI), Java primitives like hash code computation, and implementation of a subset of native methods from the JDK1.1 libraries. The Java run-time system is written in C and compiled with *gcc*. The Strata compiler itself is written in Java.

[1]Some of this description of the Strata JVM has been borrowed from the dissertation of Timothy Heil [50] because the JVM infrastructure used in this dissertation is very similar to that used by Timothy Heil.

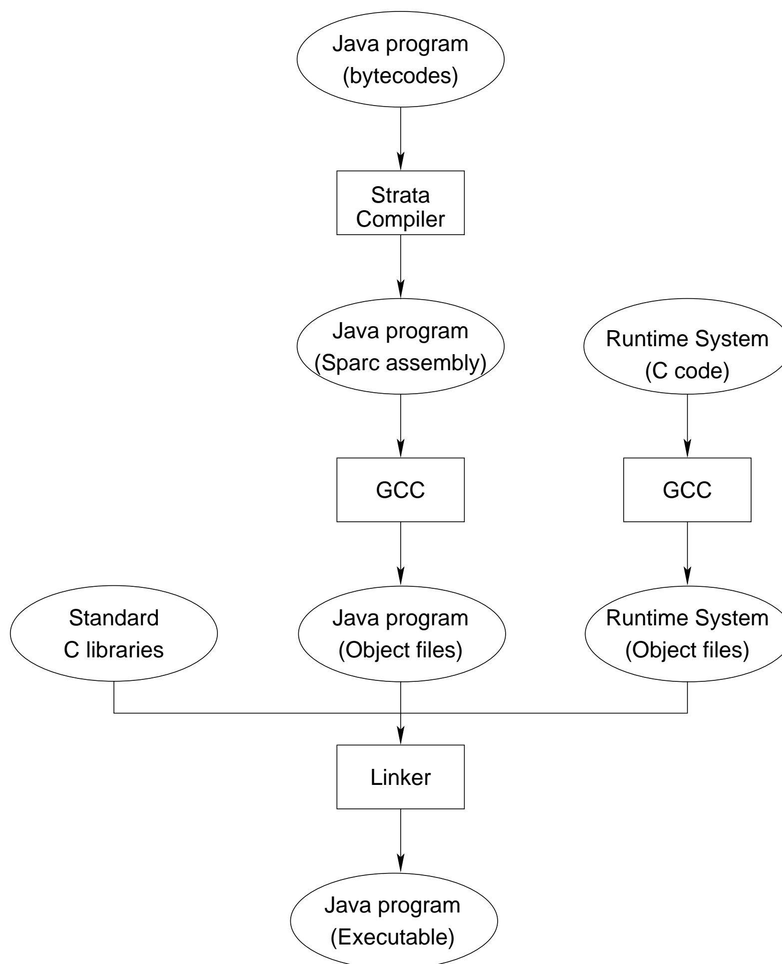

Figure 9.1. Description of the Strata JVM used to implement the specializer

9.1.1. Details of the Strata compiler

```
1.  Convert bytecodes to high-level IR
2.  Perform local optimizations
3.  Inline user-specified methods
4.  Perform Sparse Conditional Constant Propagation
5.  Convert to low-level IR
6.  Perform jump optimization
7.  Perform Global Register Allocation
8.  Layout basic blocks
9.  Generate code
```

Figure 9.2. Passes of the Strata compiler

The operation of the Strata compiler is shown in Figure 9.2. Bytecodes are first loaded and converted from the stack-based representation into a register-based high-level bytecode IR. Local (intra basic-block) optimizations are performed at this point, primarily to reduce the size of the IR. Local optimizations include copy propagation, constant propagation, common sub-expression elimination, and null-check elimination.

Function inlining is performed next. The compiler inlines only those methods that are specified by the user. In this dissertation, the inlineable methods were determined using a combination of profiling and source-code inspection.

Next, Sparse Conditional Constant Propagation (SCCP) [89] is performed after converting the IR into SSA form. The algorithm has been extended to automatically eliminate null-reference and array-bounds checks required by the Java bytecodes.

After the IR is converted back from SSA form, it is transformed into a low-level IR to make explicit the register requirements of the IR instructions. Due to both SCCP and conversion to low-level IR, a number of spurious jumps are introduced. A pass of jump target optimization eliminates many of these spurious jumps. Register allocation is performed on this low-level IR using a graph-coloring global allocator that also eliminates unnecessary copy instructions [44]. Unneeded copies are produced by the conversion out of SSA form, and by marshalling function call parameters.

Basic block layout is the final transformation before code generation. Basic block layout orders basic blocks to minimize jumps and taken branches. This algorithm is driven by heuristic-based predictions of branch directions. For instance, loop back edges are estimated to be taken 66% of the time. For many branches, such as those related to exceptions, the direction of the branch can be predicted statically with high accuracy.

While the Strata compiler also implements the Lazy Code Motion optimization, it has been turned off in our experiments because of some implementation problems in the interaction between the specializer and the lazy code motion phase. The absence of PRE-based optimizations (on register values or memory) can lead us to report higher speedups than otherwise possible. However, the effects does not affect the conclusions of our study because specialization provides benefits beyond that captured by PRE-based optimizations. First, not all eliminated instructions can be captured by PRE-based optimizations. Second, specialization unrolls loops which is one of the more significant sources of speedups in our evaluation. Therefore, the absence of PRE-based optimizations in the Strata compiler does not affect the conclusions in any significant way.

9.1.2. Emulating runtime specialization

Since Strata does not support runtime compilation, we "emulate" runtime specialization using a two-step process similar to profile-driven recompilation.

Briefly, the emulation process runs the program for N steps, creates specialized code using the profile collected in these N steps, and then continues with the execution using the specialized code in a second run.

Let `P` represent the program being specialized. In the first step, `P` is instrumented to generate profiling events for building store and object-access profiles. Let $\mathtt{P}_{\mathtt{prof}}$ represent this instrumented executable. When $\mathtt{P}_{\mathtt{prof}}$ is executed, it generates profiling events which are processed by the stratified sampler implemented in the runtime system which simulates a hardware version of the sampling

profiler. The output of the stratified sampler is used to build the store and object-access profiles needed by the specializer. Whenever the number of profiled events reaches a predefined threshold N_p (whose value is determined empirically), $\texttt{P}_{\texttt{prof}}$ is stopped and the specializer is invoked. Let t be the "time" when $\texttt{P}_{\texttt{prof}}$ is stopped. The specializer uses the runtime profiles as well as the runtime state of the interrupted program (at time t) to build scopes and generate specialized code. Note that in contrast with profile-driven recompilation, we do not run the program to completion.

In the second step, the specialized code generated during the profiling run is linked into the original program to create a new specialized program, $\texttt{P}_{\texttt{spec}}$, which is then run with the same inputs used in the profiling run. During this "specialized" run, $\texttt{P}_{\texttt{spec}}$ links in the specialized code at time t, the time when $\texttt{P}_{\texttt{prof}}$ was interrupted.

Specialization Phases

The previous discussion assumed that specialization is performed only once. However, in general, specialization is a continuous process with multiple specialization phases. During the profiling run, the specializer is triggered every N_p profiling events. This corresponds to multiple specialization phases in a dynamic optimizer. In a dynamic optimizer, the specialized code generated during each of these specialization phases is linked into P. However, in our emulation environment, we empirically select *one "best"* specialization phase and use it to generate the specialized executable, $\texttt{P}_{\texttt{spec}}$. Because of this infrastructural limitation, we cannot fully exploit all the available specialization opportunities in P. For example, different specialization opportunities might be exposed at different points in the program's execution. Exploiting both these opportunities will lead to better speedup than picking the better of these two opportunities.

Synchronizing the profiling and specialization runs

In Section 9.1.2, we mentioned that during the specialization run, the specialized code is picked up at time t, the execution point at which profiling run was interrupted. In our emulation, we use object allocation to keep track of time, i.e. whenever the program allocates an object, the clock moves forward by one tick. Therefore, time t corresponds to t objects allocated by the program. While other metrics like instruction counts, basic block counts, or method invocation counts could be used for the same purpose, we found that object allocation count is a low-overhead synchronization technique. However, because of our time-keeping methodology, the specialized code can be picked up earlier than when the program was interrupted during the profiling run. But, this synchronization error is bounded by the largest time between two object allocations. As long as the program does not have big execution phases (big relative to the entire execution of the program) where there is no object allocation, the synchronization error will be small and does not affect our reported speedup results significantly.

Detecting invalidation of specialized code

Since our infrastructure does not permit implementation of memory guarding or store monitoring schemes, our specializer implementation cannot automatically detect invalidations of specialized code. These invalidations were determined empirically as follows. If specialized memory locations change, and the associated specialized code is not invalidated, then this might manifest as a program crash or incorrect output. Using empirical investigation of the crash or incorrect output (debugging, examination of specialized code), we identified invalidations due to writes to specialized memory locations. However, this only identifies invalidations which manifest as incorrect program behavior and misses cases where the effect of the invalidation does not produce any visible difference in program behavior. This is a shortcoming of our current evaluation methodology and can be fixed in a runtime implementation of a specializer.

9.1.3. Benchmarks

We conduct our experimental runs on a lightly loaded UltraSparc10 processor with 256M RAM and use a 32M heap for object allocation. Table 9.1 shows the programs used to evaluate the specializer. The table provides a short description of the program along with the input to each benchmark. `jess` and `raytrace` are programs from the SPECjvm98 benchmark suite used to evaluate JVMs.

Program	**Description**	**Input**
`dotproduct`	*Computes dot product of two vectors*	Two 100-element vectors One vector is 75% zero-filled
`query`	*Searches database entries*	query with 21 comparisons
`interpreter`	*Interpreter microbenchmark*	Bubblesort of a 50-elt array
`convolve`	*Image processing kernel*	5x5 random convolution matrix (12 zeros, 8 ones, 4 negative ones) and a 860x550 gif image
`jscheme`	*Scheme interpreter*	Simple Scheme Partial Evaluator
`raytrace`	*SPECjvm98 benchmark*	300 300 input/time-test.model
`jess`	*SPECjvm98 benchmark*	100

Table 9.1. Benchmarks used to evaluate the specialization technique

`query` and `dotproduct` are microbenchmarks used to evaluate DyC [47] that we converted to Java. `interpreter` is a tiny interpreter microbenchmark. `convolve` has been carved out of *ImageJ*, a publicly available Java image processing toolkit [2]. `jscheme` is a publicly available Scheme interpreter in Java [74]. Note that the input to `raytrace` benchmark is different from the input configurations of the SPECjvm98 suite. We provided a different input to the benchmark to run the program for twice as long as the longest-running SPECjvm98 input. This longer-running input was provided to get the program to run for more than 30s to allow for the relative specialization overheads to be lower – the longer a program runs, the greater the benefit of specialization.

Table 9.2 shows the sampling rate (for profiling) as well as the specialization trigger threshold used in generating the specialized versions. For all the benchmarks, the stratified sampler used a table with 2K counters. For the microbenchmarks, the sampling rates and the length of the spe-

Program	Profile Sampling Rate (1/n)	Specialization Trigger Threshold (# profiling events)
dotproduct	64	64K
query	64	64K
interpreter	64	64K
convolve	256	4M
jscheme	256	0.5M
raytrace	256	4M
jess	256	4M

Table 9.2. Specialization parameters (sampling rate and specialization trigger threshold) for the benchmarks

cialization phases are smaller than the bigger benchmarks. For `jscheme`, we have used a smaller specialization phase than the rest because it enables earlier specialization (and better speedups) than with a longer specialization interval. In a transparent specializer, these parameters (sampling rate and length of a profiling phase) have to be determined automatically. While this dissertation has not studied techniques for doing this, we present some possible solutions for achieving this.

A simple way of determining the sampling rate would be to use a default sampling rate (for example, 1/256) and vary it based on the number of messages generated by the profiler within a fixed time. If the number of messages are higher than an empirical threshold, the sampling rate can be reduced, and if lower, the sampling rate can be increased.

A straightforward way of determining the length of a profiling phase would be based on the size of the working set. The profiling phase should be long enough so that the number of samples collected during the phase cover a minimum number of executions of the working set. Therefore, with small working sets, the profiling phase will be short, and with larger working sets, the profiling phase will be long.

Baseline measurements: Comparison of Strata with Hotspot Client VM

Table 9.3 shows a comparison of Strata-generated code with Hotspot generated code. Note that Hotspot is a state-of-the-art dynamic optimizer which has been optimized for the Sparc architecture.

Program	Compilation Time (Strata) (secs)	Execution time	
		Strata VM (secs)	Java Hotspot Client VM (secs)
dotproduct	5.8	10.9	11.1
query	7.6	11.4	8.7
interpreter	7.9	23.2	24.4
convolve	18.3	37.6	47.0
jscheme	36.6	48.7	22.7
raytrace	45.9	41.1	82.5
jess	102.6	49.1	27.0

Table 9.3. Baseline times for all benchmarks: time to generate Sparc V8 assembly using Strata, time to execute the program with the Strata VM, and time to execute the program with the Hotspot Client VM

We make this comparison to demonstrate that specialization is not optimizing over an inefficient compiler. This is important because a number of commonly-used optimizations can speed up program execution. Comparing the execution times of the different Java programs, we see that the execution times of Hotspot and Strata are similar for most programs except for `jscheme` and `raytrace`. For `jscheme`, Strata-generated code is 2X slower than that generated by Hotspot. For `raytrace`, Strata-generated code is 2X faster than the code generated by Hotspot. We interpret these results to signify that Strata generates code as good as would be generated by a real dynamic optimizer.

The Strata compiler generates code that is not optimized for the Sparc architecture. For example, it does not fill up delay slots. Combined with lazy code motion (or some other more efficient version of partial redundancy elimination algorithm suitable for a dynamic optimizer), the performance of Strata-generated code can be improved. However, these improvements might not significantly impact the specialization benefits reported in this dissertation because some of the code improvements carry over to the code generated by the specializer.

For the couple of cases when Hotspot runs slower than Strata, this might be due to the compilation overheads that Hotspot incurs at runtime.

More pertinent to the results that we will present later in this chapter are the compilation over-

heads. For specialization to be beneficial, compilation overheads should be small. By comparing Strata's compilation costs to Hotspot's dynamic compilation costs, we want to show that Strata's high compilation costs is not representative of runtime compilation overheads.

The Hotspot VM is a dynamic optimization system. It does as well (or better) than the Strata static compilation system despite incurring compilation overheads at runtime. This underscores the fact that dynamic optimization systems have well-engineered compilation systems designed for speed and low runtime overheads. In contrast, the Strata compiler has not been designed for fast optimization and code generation. One reason why Hotspot has a low-compilation cost when compared to Strata is because Hotspot probably only optimizes hot methods. However, Table 9.3 shows that Hotspot does much better on `query`, a simple micro-benchmark, which has just a few methods to be compiled. Hotspot compiles and executes the program much faster than it takes Strata to compile and execute the program. Therefore, Strata's high compilation costs are probably an artifact of the design of the Strata VM and do not reflect the true costs of dynamic compilation.

Having presented the research infrastructure used to implement the specializer and the methodology for evaluating it, we now present results of our experimental evaluation in the following sections. We first present an evaluation of the store profile: its time and space overheads and its ability to detect invariance. Next, we present an evaluation of the time overheads of scope building, and creating specialized versions. Finally, we present the speedups from specialization and discuss the results.

9.2. Evaluating the store and object-access profiles

In this section, we present an evaluation of the store and object-access profiles to understand their suitability for use within a dynamic specializer. First, we present a discussion of the time and space overheads of the profiles. Then, we discuss the accuracy of the store profile in determining runtime invariance.

9.2.1. Time and space overheads

Our evaluation shows that the time overheads of profiling will be between 5% and 10% and that the space overheads of profiling will be under 1MB for most applications. We now present a discussion of these results next.

In our implementation, we use the stratified sampling based profiler presented in Chapter 4 for collecting the store and object-access profiles. We do not perform a separate evaluation to determine the time overheads to collect these profiles. Instead, we rely on the earlier evaluation presented in Chapter 4 where we showed that a stratified-sampling based profiler can collect a load value profile with overheads a little over 5%; with a two-level compression scheme, the overheads are a little over 4% (Figure 4.12). The object-access profile will have a similar time overheads because in Java, object and array accesses are primary source of load instructions (others are virtual table accesses, accesses in the runtime system, stack refill, restores, etc). The store profile will have much smaller overheads than 5% because a program executes far fewer stores than loads. Thus, the overheads of simultaneously collecting both these profiles will definitely be less than 10%, and not significantly more than 5%.

Let us now examine the memory requirements of the object-access profile. The object-access profile had a maximum footprint of 3MB for `raytrace`. For `jess`, it is under 2MB, and for the other benchmarks, it is under 250KB. This is not a significant memory overhead. Some of this overhead is an artifact of the Strata implementation and can be reduced significantly with better engineering. We now describe one optimization which can lead to a reduction in space overheads of the object-access profile. Right now, the specializer retains profile records for all instructions. However, for some instructions (those that access "too many" objects), there is no use retaining this information since the specializer will not specialize that instruction. By freeing profile records for these instructions, the space overhead of the object access profile can be reduced considerably without impacting its accuracy for the purpose of specialization.

Let us now examine the memory requirements of the store profile. For the sampling rates and specialization trigger thresholds shown in Table 9.2, `convolve` had the biggest memory footprint at 223KB. This is small compared to memory footprints of applications today – with a application memory footprint of 20MB, 223KB is around 1% space overhead. This space overhead can be further reduced by a better engineered implementation; like most parts of our specializer, the profile representation has not been engineered for efficiency.

Thus, the combined space overhead of the store and object access profiles is expected to be under 1MB which is an acceptable space overhead for today's applications with much larger memory footprints.

9.2.2. Accuracy of the store profile

In the rest of this section, we show that a store profile enables accurate invariance detection. Accurate invariance detection is important because if a profile fails to identify that a memory location is actually variant, then specialization may need to be frequently invalidated, reducing the effectiveness of specialization. For our benchmarks, we encountered invalidations only in a couple of cases, one for the `interpreter` microbenchmark and the other for the `jscheme` benchmark. In both these cases, the invalidation was rare and in fact beneficial, because it enables more aggressive specialization. We discuss these scenarios next.

There are two sources of potential inaccuracy in the store profile. The first, and fundamental, potential source of inaccuracy is due to the fact that the invariance detection is based on execution history, i.e. the store profile predicts the invariance of a memory location based on the absence of any stores to the location in the past. However, clearly, this is not a guarantee of invariance – the location could be written in the future. The second source of inaccuracy arises because the stores are sampled. Therefore, many stores do not show in the sample and adds to the first source of inaccuracy – more locations are predicted to be invariant than if every store were monitored.

Invalidation in the `interpreter` microbenchmark:

We now show an invalidation scenario in the `interpreter` benchmark and show that it is rare and can, in fact, enable more aggressive specialization. This invalidation is a result of the first kind of inaccuracy – where history-based prediction is not a guarantee of invariance.

A snippet of the benchmark code is shown in Figure 9.3. Looking at the figure, we see that it accesses three different arrays: `pgm`, `reg`, and `mem`. These arrays are the three potential sources of invariance within the interpreter. We now discuss the scenario where elements of the `reg` array are modified after they are used for specializing the method.

```
int Interpret(Instruction[] pgm, int pc)
{
   for (; ;) {
      Instruction i = pgm[pc]; int newpc = pc+1;
      switch(i.opc) {
         case LI    : reg[i.rt] = i.imm; break;
         case GOTO  : newpc = i.imm; break;
         case SUBI  : reg[i.rt] = reg[i.rs] - i.imm; break;
         case LD    : reg[i.rd] = mem[reg[i.rs] + reg[i.rt]]; break;
         ....
      }
      pc = newpc;
   }
}
```

Figure 9.3. Interpreter Program

First, `pgm`, the instruction array that contains the input program will be invariant throughout the program's execution (assuming no self-modifying code). The specializer commonly creates multiple specialized versions of the interpreter loop iteration – one for each frequently interpreted instruction of the source program. These specialized code rely on the invariant memory locations that are identified by the store profile – specifically, elements of the `pgm` array, but also elements of the `reg` array as we discuss next.

Second, the interpreter uses the integer array `reg`, to record the register values of the source pro-

gram. Thus, a register access `r1` in the source program corresponds to the memory access `reg[1]` in the interpreter. Therefore, if the source program stores constant values in its registers, these constants manifest as invariant elements in the `reg` interpreter array.

Third, the interpreter uses the `mem` array to store the memory contents of the source program. Therefore, if the source program has invariant memory, it directly manifests as invariant locations in the `mem` interpreter array.

While the `pgm` array is invariant throughout the program's execution, this is not necessarily true about the `reg` and `mem` arrays. Figure 9.3 shows that these arrays are modified during interpretation. However, individual elements in these arrays can be invariant for extended periods during interpretation. If the specializer exploits these temporally semi-invariant values in the `reg` and `mem` arrays, it can lead to the invalidation of specialized code when these elements are modified. This scenario is seen when interpreting the bubble-sort program shown in Figure 9.4.

```
void BubbleSort(int[] a)
{
   int tmp;
   int n = a.length;
   for (int i = 0; i < n; i++) {
      for (int j = i+1; j < n; j++)
         if (a[i] > a[j]) tmp = a[i]; a[i] = a[j]; a[j] = tmp;
   }
}
```

Figure 9.4. Bubble Sort input program to the interpreter

The bubble-sort source program uses indices `i` and `j` for its nested (sorting) loops. In the compiled version of bubble-sort, let us suppose that these indices are in registers `r1` and `r2` respectively. Within the interpreter, these indices will be in the interpreter's register array at `reg[1]` and `reg[2]`.

While interpreting the bubble-sort program, the interpreter spends most of its time in bubble-sort's inner loop. This interpretation of the inner loop might span multiple specialization phases

depending on the size of the array being sorted. During this time, the value of the outer-loop index (`i`) does not change. Therefore, there will be no stores to the interpreter memory location `reg[1]` in this time. Consequently, the specializer optimistically assumes that `reg[1]` is invariant and uses the value of `reg[1]` in generating the specialized code. However, when the outer-loop index `i` is eventually updated, all specialized code that used the value of `reg[1]` have to be invalidated. For the specialized code generated by the specializer, only one of the eight specialized versions have to be invalidated – that corresponding to the use of the value of `reg[1]`.

Since our current specializer implementation does not support invalidation, for the purposes of evaluation, we changed the input set so that invalidation is avoided. We reduced the input array size to size 50 (shown in Table 9.1) so that the index value change is recorded by the store profile. All our results for the `interpreter` benchmark are based on this small input size.

Thus, in general, while invalidation signals bad store-profile based predictions of memory invariance, in cases such as this example, it can be beneficial since it enables more aggressive specialization. Within a dynamic optimization environment, this semi-invariance of `i` can be exploited by respecializing the loop iteration whenever `i` is modified. Exploiting such respecialization opportunities is left for future work.

Invalidation in the `jscheme` benchmark

`jscheme` is an interpreter for the functional language, Scheme. The normal operation of the interpreter is to process user input provided at the interpreter prompt. In this mode, the user defines variables and functions and/or loads existing definitions from a file. These definitions are stored in a *lookup environment* which maps names of variables and functions to their definitions. During interpretation, lookup environments are consulted to fetch the values that variable and function names are bound to.

In a batch-processing environment, where all input is provided at the beginning of the program,

the contents of the lookup environment is invariant or rarely changes. In such a scenario, the specializer creates a specialized version of the lookup function in which the traversal of the environment is eliminated. However, when new definitions are added, the lookup environment changes and the specialized version has to be invalidated.

In the original experiment, input to the interpreter was presented interactively. However, this invalidates specialized code at every instance when a new definition is provided by the user. Just like the `interpreter` microbenchmark, these invalidations signal a need for respecialization and is not an erroneous detection of invariance by the store profile.

In our evaluation, we got around this scenario by converting the interactive input into a batch input. All results for the `jscheme` benchmark are reported for this input.

9.2.3. Conclusion

Based on our evaluation, we conclude that the store profile accurately identifies invariant data. In the cases where specialized code was invalidated, we discovered that this resulted from temporally semi-invariant data and can potentially enable more aggressive specialization. When combined with the low time and space overheads, we are led to conclude that store-profile based invariance detection is a useful technique for transparent runtime specializers.

9.3. Runtime compilation costs

In this section, we present the runtime compilation costs of our scope-building and specialization algorithms. In Section 9.1.2, we described the emulation process used to implement runtime specialization. We rely on emulation because the Strata VM does not support runtime compilation. This emulation, unfortunately, prevents us from directly measuring the overhead our specializer would incur when implemented in a dynamic optimizer. Therefore, we present *estimates* of runtime compilation costs in the following sections.

9.3.1. Scope-Building Algorithm

In this dissertation, we estimate the compilation cost of the scope-building algorithm by comparing its execution time with that of a SCCP constant propagator which is an extremely efficient linear-time algorithm. If the two speeds are comparable, then this would demonstrate that the scope-building algorithm can be efficiently implemented in a runtime setting since SCCP is a basic constant propagation algorithm likely available in most dynamic optimizers.

Program	**Num. Scopes**	**Scope Building Costs**	
		t_{Scopes}/t_{SCCP} **(min, max)**	**Worst absolute time (milli-secs)**
`dotproduct`	1	(0.6, 0.6)	50
`query`	1	(1.2, 1.2)	150
`interpreter`	1	(1.5, 1.5)	170
`convolve`	1	(0.2, 0.2)	50
`jscheme`	1	(3.3, 3.3)	100
`raytrace`	6	(3.0, 6.1)	1040
`jess`	5	(2.2, 6.7)	270

Table 9.4. Scope-building costs relative to SCCP

In Table 9.4, we show the compilation overheads for the scope-building algorithm relative to the standard SCCP algorithm. The second column shows the number of beneficial scopes that were identified.[2] The third column shows the ratio t_{Scopes}/t_{SCCP}, where t_{Scopes} is the time taken by the scope-building algorithm to build scopes for hot methods, and t_{SCCP} is the time taken by the SCCP algorithm to process the same methods. This ratio is only presented for the methods for which beneficial scopes were identified. The fourth column shows the maximum time taken by the scope building algorithm for any method.

The table shows that the scope building algorithm runs quite efficiently for the microbenchmarks and the `convolve` benchmark. For the more larger applications `jscheme`, `raytrace`,

[2]For `jscheme`, `dotproduct`, `query`, and `interpreter`, the scope-building algorithm identified more than one scope. However, many of the scopes are only active during the initialization phase (reading input files, building data structures) and do not contribute significantly towards either the cost of specialization or towards the benefit from specialization and therefore have been eliminated.

and `jess`, the scope-building algorithm runs within 3X the speed of the SCCP algorithm in the best case and within 7X in the worst case. In terms of absolute times, the fourth column shows that the absolute time are well under a second except for the `OctNode.Intersect` method in the `raytrace` benchmark. This method is one of the larger methods in the application.

These compilation times can be reduced with a better engineered implementation. In addition, a part of the scope-building cost is an artifact of our emulation methodology. During the emulation, the specializer incurs interprocess communication overheads while accessing the application runtime state via the profiler process.

Even with our relatively poorly-engineered implementation, the per-method average absolute scope-building cost for *all* hot methods (not just for methods with beneficial scopes) processed by the specializer across all benchmarks was only 137 milliseconds. The cost exceeded 500ms for only 4% of all processed methods. This is a very encouraging result because such low costs can enable a runtime implementation of an automatic, profile-driven scope-building algorithm.

This result is encouraging for an additional reason; the scope-building algorithm can run in the background all the time and can continue processing hot methods periodically and trigger specialization whenever a beneficial scope is identified. The actual cost of specialization (which is much higher) is incurred only for beneficial scopes. This result enables a runtime implementation of a specializer within a dynamic optimizer which is active for the entire execution of a program.

9.3.2. Specialization Algorithm

In this section, we present an evaluation of the costs of generating specialized code for the scopes identified by the scope-building algorithm.

In Table 9.5, we show the cost of specializing a method relative to regular compilation costs for the same method. This specialization cost includes the costs of scope building, creating multiple specialized versions of beneficial scopes, global register allocation, basic block layout, and code

Program	Num. Scopes	Versions	Loops	Total unrolled iterations	Specialization costs	
					t_{Spec}/t_{NoSpec} (min, max)	Total time (secs)
dotproduct	1	1	1	100	(2.6, 2.6)	4.0
query	1	1	1	22	(1.3, 1.3)	2.7
interpreter	1	8	0	0	(1.7, 1.7)	3.1
convolve	1	1	2 (nested)	25 (5 x 5)	(10.6, 10.6)	13.2
jscheme	1	1	1	235	(28.9, 28.9)	13.0
raytrace	6	70	1	153	(1.3, 33.0)	39.4
jess	5	28	5	157	(2.5, 37.0)	26.9

Table 9.5. Total specialization costs relative to regular compilation

generation. For benchmarks with multiple scopes, the sixth column also shows the best and worst ratios. The last column shows the total absolute time for specialization. The table also shows the number of specialized versions created, number of loops unrolled, and the total number of unrolled iterations across all scopes.

For `dotproduct`, `query`, `convolve`, and `jscheme`, the specializer identifies only one beneficial scope with only one specialized version. For these benchmarks, it can be verified that the higher relative compilation costs are directly correlated with loop unrolling. However, except for `convolve`, the compilation ratio is about an order of magnitude smaller than the unrolling factor. This shows that the SCCP-based specialization algorithm can efficiently unroll and specialize the selected specialization scopes.

For `interpreter`, the compilation ratio (1.7) is smaller than the number of generated specialized versions (8). This strengthens the conclusion that SCCP-based specialization is efficient when deployed.

For `raytrace`, the largest compilation cost was incurred for generating specialized versions of the `FindTreeNode` method shown in Figure 1.1. The specializer creates 53 versions of this method and many of them had loops unrolled upto 8 times. For this method, the relative compilation cost is 33X the cost of regular compilation.

For `jess`, the largest compilation cost was incurred for generating 12 specialized versions of

the `ValueVector.equals` method which had a loop unrolled in each of them for a total of 84 unrolled iterations. For this method, the relative compilation cost is 37X the cost of regular compilation.

For both these SPECjvm98 benchmarks, the relative compilation costs follow a similar trend as those for the smaller benchmarks. The costs are much smaller when compared to the amount of code duplication (through unrolling and multiple specialized versions).

Reducing specialization costs

For all these benchmarks, despite the favorable compilation ratios, the absolute compilation times seem quite high. Some of this high compilation time comes from the design of the Strata compiler which has not been engineered for fast compilation, as we noted in Section 9.1.3. Some of it also comes from the emulation methodology which requires inter-process communication with the profiling process. Therefore, we believe that specialization times can be reduced within a well-engineered dynamic optimizer. Besides this, compilation costs can be reduced further by using multiple threads of control that share the processor with the application [11, 60, 67]. Thus, we conclude that SCCP-based specialization (with loop unrolling) can be implemented efficiently within a dynamic optimizer.

9.3.3. Summary

Based on our evaluation of the compilation costs of profile-driven automatic scope building (Chapter 5) and SCCP-based specialization (Chapter 6), it appears that these algorithms can be efficiently implemented within a dynamic optimizer.

The automatic scope building algorithm is especially promising since it is quite efficient and is comparable in speed with a SCCP algorithm. Our evaluation shows that the average cost of the scope building algorithm is less than 150ms. This cost can be further reduced with a better-engineered

implementation.

While the SCCP-specialized algorithm incurred high compilation costs relative to regular compilation, these higher costs correlate directly with the code duplication performed by the specializer. Our evaluation shows that the extra costs lines up quite favorably when compared to the code duplication performed by the specializer.

With a well-engineered implementation of a dynamic optimizer that uses multiple concurrent compilation threads, specialization costs can probably be reduced sufficiently enough to enable the implementation of a specializer at runtime.

9.4. Speedup estimates without including specialization costs

In this section, we present the speedups due to specialization and discuss the results. These speedups *do not* include overheads for profiling, scope building, specialization, or software guards for invalidation. We obtained these numbers by taking the average of 8 runs out of 10 after eliminating the best and worst runs. In order to minimize variation due to code layout effects, we used the same executable to time the specialized and unspecialized runs using a command-line flag passed to the Strata runtime system.

Program	**Asymptotic Speedup (without overheads)**	t_{NoSpec} **(in secs)**	t_{Spec} **(in secs)**
`dotproduct`	5.1	10.7	2.1
`query`	3.7	11.0	3.0
`interpreter`	1.7	23.7	14.1
`convolve`	2.0	36.1	17.5
`jscheme`	3.6	52.1	14.3
`raytrace`	1.07	49.3	46.2
`jess`	1.03	47.5	46.1

Table 9.6. Speedups due to specialization: ratio and absolute times

Table 9.6 shows the results of specialization for all the benchmarks. The second column shows the asymptotic speedups. The third and fourth columns show the absolute execution time (in seconds)

for the unspecialized and specialized code versions respectively (timed using the same executable as explained earlier).

On the microbenchmarks, the image-processing kernel, and the Scheme interpreter, specialization yields good speedups. The speedup numbers shown in this table should be interpreted that the specialization technique (without regard to whether it is done at compile-time or run-time) is powerful enough to identify specialization opportunities and exploit them. However, this table does not tell us whether the technique is good enough to be employed at runtime since it does not show us the overheads. These numbers are presented later on in this section.

`dotproduct` and `query` are microbenchmarks used to evaluate the DyC specializer and it is encouraging that our specializer could automatically identify and exploit the same specialization opportunities with low overheads.

On the `interpreter` benchmark, while the speedup of 1.7 is encouraging, there is a lot of room for improvement. In Section 10.2, we examine some limitations of our specialization approach and argue that the limitations are an artifact of our specialization model, which can be, in principle, easily removed.

The impressive speedup on `jscheme` is especially encouraging since it reflects the potential of runtime specialization in speeding up interpreters, especially because it is not a toy interpreter. All the benefit for this benchmark comes from the specialization of the `lookup` method shown in Figure 9.6. To specialize this method, we had to make minor source-code modifications to overcome a limitation of our intermediate representation. This is explained further in greater detail in Section 9.4.1. Preliminary investigations of the source code of the interpreter reveal that there are further specialization opportunities in the interpreter which can potentially be exploited. These opportunities come from the `eval` method – the core evaluation routine of the Scheme interpreter (similar to the core interpretation loop in interpreters of imperative languages, like the `interpreter` benchmark shown in Figure 9.3). Further investigation is left for future work.

The larger SpecJVM98 benchmarks show a more disappointing performance, mainly because they are not easily specializeable, but also because of the limitations of our specialization system. Most of the benefit in `raytrace` comes from the `FindTreeNode` method shown in Figure 1.1. The specializer specializes away the pointer traversal of the octree for the commonly referenced octree nodes. The application spends about 25% of its time in this method and the speedup of 1.07 is a good result for speedups coming primarily from this method. While the specializer identified and created specialized code for scopes in other methods, the resulting benefit was minimal. The reasons for this have not been investigated at this time and is left for future work.

For `jess`, most of the benefit comes from creating specialized versions of the `Node2.runTests` method. Since `jess` is an expert system shell, ideally, the specializer should do better on this benchmark. However, one of the reasons for the poor specialization is because of our restrictions on lookup keys. `jess` performs a number of equality comparisons which can be exploited if the specialization model supported 2-element keys. It should be noted that multi-variable scope-building algorithms exist (ex: Calpa [71]), and some of the ideas from existing work could be adapted to extend the scope-building algorithm studied in this dissertation.

9.4.1. Caveats

Discrepancies in baseline execution times due to emulation overheads

In this section, we examine the discrepancies in the run times between the regular baseline run of an executable (shown in Table 9.3 and the unspecialized run of the specialized executable (shown in Table 9.6). Ideally, these two run times should be the same. However, in our system, they will be different for two reasons: due to emulation overheads in the JVM runtime system of the specialized executable, as well as due to code layout effects due to presence of the extra specialized code in the specialized executable. Some emulation overheads are incurred even though specialized code is not executed, and the code layout effects will also manifest themselves even though specialized code is

not executed.

By comparing the execution times for the unspecialized version shown in Table 9.6 with the baseline execution times shown in Table 9.3, we notice that except for `raytrace`, the times are with 4% of each other. For `dotproduct`, `query`, `convolve` and `jess`, the baseline times are slightly slower. For `interpreter` and `jscheme`, the baseline times are slightly faster. However, the variations are small enough (within 4%) that this has not been investigated closely, especially since this is just an artifact of our evaluation methodology.

However, the specialized executable (without exercising specialization) for `raytrace` runs about 20% slower than the baseline version. To investigate this discrepancy, we created a new executable for `raytrace` which has the JVM runtime with the emulation system used for specialization, but without linking in any specialized code. This new executable takes 46.3 seconds which is slower than the original baseline code – this could be due to the emulation overheads in the runtime system. However, the unspecialized code still runs 6% slower than with this new executable (49.3 secs vs 46.3 secs). This is very likely a result of i-cache effects due to the specialized code. The executable size increases by about 7% for `raytrace` which is much higher than all the other benchmarks (about 2 to 3% for the others). Even though the extra 7% specialized code is never referenced in the unspecialized run, the code layout might influence the i-cache miss rates. A more precise investigation requires access to a cache simulator or a micro-architectural simulator for the UltraSparc architecture.

Source-code change for `jscheme`

`jscheme` benefits significantly from specialization. The specializer identifies the `Environment.lookup` method for specialization and completely unrolls and specializes the loop that searches the lookup environment for the value that a variable is bound to.

In order to enable the scope-building algorithm to identify the unrolling opportunity in this scope,

```
Object lookup(String sym) {
   Object vs = vars, cs = vals;
   while (vs != null) {
1.    x = (vs instanceof Pair) ?((Pair)vs).first :null;
      if (x == sym) {
         return first(cs);
      } else if (vs == sym) {
         return cs;
      } else {
2.       vs = (vs instanceof Pair) ?((Pair)vs).rest :null;
         cs = rest(cs);
      }
   }
   ....
}
```

Figure 9.5. Original lookup method in `jscheme`

```
Object lookup(String sym) {
   Object vs = vars, cs = vals;
   while (vs != null) {
      if ((vs instanceof Pair) && (((Pair)vs).first == sym))
         return first(cs);
      else if (vs == sym)
         return cs;
      else if (vs instanceof Pair)
         vs = ((Pair)vs).rest;
      else
         break;
      cs = rest(cs);
   }
   ....
}

/* Generated specialized code for the lookup method */
Object lookup_SPEC(String sym) {
   if ((a1 == sym)) return c1;  /* a1, a2, c1, c2, ... are constants */
   else if (b1 == sym) return d1;
   if ((a2 == sym)) return c2;
   else if (b2 == sym) return d2;
   ....
}
```

Figure 9.6. Modified lookup method in `jscheme` and the corresponding specialized code

we made some source-code modifications shown in Figure 9.5 and Figure 9.6. The specialized code generated after this transformation is shown in Figure 9.6. The changes stem from lines 1 and 2 in Figure 9.5 both of which get encoded into ϕ-nodes in the SSA graph. As a result, the values of `x` in Line 1 and the value of `vs` in Line 2 come from these ϕ-nodes. Our current scope-building algorithm handles these ϕ-nodes conservatively as we explain now. Referring back to the transfer function for ϕ-nodes in Table 5.2, we notice that this function computes a meet of the incoming values. As a result, the values of `x` and `vs` go to $\perp$ during the analysis. This forces the algorithm to end the scope at Line 1 and Line 2. However, on examining the code in Line 1, we see that the value of `x` can be determined at specialization time if we know the value of `vs`. This is because the branch is computable at specialization time. The same holds true for Line 2.

This scenario can be handled automatically (without code modification) in one of two ways. One approach would be model the dataflow analysis shown in Table 5.2 along the lines of SCCP – where values are evaluated conditionally (only after the controlling branch has been evaluated).

A second approach would be use a gated-SSA form which links up the ϕ-node with its controlling branch. This will enable control-flow information to flow from the branch to the output of the ϕ-node. If the specializer uses this representation, the algorithm can automatically eliminate the ϕ and continue growing the scope.

In the absence of either of these approaches, we modified the method as shown in Figure 9.6 to eliminate both the ϕ-nodes. With this modification, the algorithm identifies the entire loop as the specialization scope and marks the loop for unrolling. This source-code modification is similar in spirit to *binding-time improvements* in offline partial evaluators [56]. As a result of the modification, the specializer was able to unroll the loop and create specialized code that resembles the code shown in Figure 9.6.

Program	t_{NoSpec} **(in secs)**	t_{cost} **(in secs)**	t_{Spec} **(in secs)**	New speedups $t_{NoSpec}/(t_{cost} + t_{Spec})$
`dotproduct`	10.7	4.0	2.1	1.7
`query`	11.0	2.7	3.0	1.9
`interpreter`	23.7	3.1	14.1	1.4
`convolve`	36.1	13.2	17.5	1.2
`jscheme`	52.1	13.0	14.3	1.9
`raytrace`	46.9	39.4	43.7	0.56
`jess`	48.0	26.9	46.1	0.66

Table 9.7. Consolidating results from Table 9.5 and Table 9.6: the lost column shows speedups when the compilation costs from Table 9.5 are included.

9.5. Speedup estimates after including specialization costs

Table 9.7 consolidates the results from Table 9.5 and Table 9.6. Note that the specialization costs shown in Table 9.5 includes the cost of scope building, the cost of creating specialized versions of beneficial scopes, the cost of other optimization passes shown in Figure 9.2, and code generation. We add this specialization cost to the execution time of the specialized version and compute new speedup numbers which are shown in column 5. The table shows that even with our poorly-engineered specializer, the resulting benefit from specialization is non-trivial for five of the benchmarks. For `raytrace` and `jess`, there is significant slowdown. Firstly, the improvement due to specialization is minimal. In addition, the specialization costs are significant. Both of these can be addressed: the improvements due to specialization can be increased, and the overheads due to specialization can be reduced. In addition, when the specializer is implemented at runtime, the cost-benefit model presented in Section 5.3 would have to be improved to include the costs of runtime specialization – such a cost-benefit model will be able to prevent large slowdowns where there is not much benefit.

When seen in this new light, these speedup numbers look quite encouraging. But, they should be consumed with “a pinch of salt” because these numbers only include specialization costs from a single specialization phase, and still do not include profiling and invalidation-support costs. In addition, they do not incorporate costs involved in processing other “non-beneficial” methods, and

other dynamic optimization overheads which we might not be aware of (in the absence of a real dynamic optimization environment).

Nevertheless, these numbers are promising for the following reasons. The specialization overheads shown in column 3 can be reduced further with better engineering as we discussed in Section 9.3.3. In addition, there exist known limitations of our specializer, which if fixed, can provide better results than presented in this dissertation.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

Recent times have seen the advent and development of a number of dynamic optimization systems [3, 15, 21, 25, 35, 53, 57, 67], some of which have been spurred by the need to optimize execution of Java programs. Program development in Java is probably indicative of the nature of component-based software development today: object-oriented programming, reliance on dynamic linking techniques, and use of generic software libraries.

Dynamic optimizers, by virtue of having the ability to monitor program behavior and having access to the program's runtime state, can tailor program optimizations to the program's runtime context (runtime state, hardware characteristics). However, existing optimizers implement control-flow optimizations and leave unexploited data-specific optimizations.

Program specialization can exploit these data-specific optimizations within dynamic optimizers by eliminating computation that depend on runtime constants. This dissertation explored the possibility of implementing program specialization transparently, at runtime, within dynamic optimizers.

10.1. Summary of contributions

Program specialization is a powerful optimization technique that relies on powerful program analysis and transformations. Even in its compile-time incarnation, specialization techniques for imperative languages [8, 31, 47, 70] are not easily implemented due to their conceptual and implementation complexity. This is so even when the user tells the specializer what is invariant, and what to specialize. Within the context of transparent dynamic optimizers, a newer challenge pops up – the entire process has to be automated with low runtime costs. Earlier, where the user or compile-time program analysis provided the specializer with the information about what is invariant and what to specialize, now, the specializer has to find all this information on its own at runtime.

In this light, the approach taken in this dissertation is that of simplicity: "what will happen if

all the pieces of the puzzle are kept as simple as possible?" This approach is reflected in the various pieces of the dissertation: specialization model, runtime profiling, invariance detection, automatic scope building, specialization, dispatching mechanisms, and store monitoring techniques. With respect to these various pieces of the problem, this dissertation makes the following contributions:

- *Stratified-sampling based runtime profiling:* We showed that rather than design complex hardware to collect low-overhead profiles, using the samples provided by a simple hardware stratified sampler, the software can build a variety of runtime profiles with low overheads and high accuracy. Our evaluation shows that even a very demanding profiling application like load value profiling has runtime overheads of under 6%.

- *Store-profile based invariance detection:* We showed that, by profiling program stores, it is possible to detect invariance in data structures at fine granularities. This approach is much simpler and also more powerful than program analysis techniques that detect data structure invariance. In addition, we also showed that this approach also makes it possible to detect temporal semi-invariance in data structures.

- *Profile-driven automatic scope building:* We showed that using information provided by the store and object-access profiles, and by employing several simplifying heuristics, suitable specialization scopes can be automatically identified by a data-flow analysis algorithm. Our evaluation shows that even our unoptimized implementation can identify specialization scopes efficiently. The speed of the scope-building algorithm is comparable to the linear-time SCCP algorithm. Our evaluation showed that on an UltraSparc10 processor, on average, the scope-finding algorithm takes under 150ms to process a hot method.

- *SCCP-based specialization:* We showed that a Sparse Conditional Constant Propagator can be turned into an effective specializer (based on the online partial evaluation strategy) by relying on a store profile to eliminate invariant memory references. In addition, we showed that SCCP

can be extended to unroll loops and specialize them. Our evaluation shows that while the specialization overheads are non-trivial, the overheads are directly correlated with the amount of code duplication (in the form of multiple specialized versions, and loop unrolling). Better-engineered implementations can reduce these overheads further.

- *Pseudo-method based dispatching to specialized code:* We showed that the restricted specialization model can be exploited to implement low-overhead dispatching to specialized code. This technique treats specialization scopes like methods and uses object method tables to implement dispatching.

- *Techniques for detecting violations of memory invariance:* We showed that violations of memory invariance can be detected using several techniques. The simplest technique relies on fine-grained memory protection that has been proposed recently [90]. In addition, we proposed several software techniques for detecting memory violations. We showed that in certain cases, store monitors can be combined with garbage collector write barriers which eliminates the overheads of store monitors in these cases. However, these proposed techniques need implementation and evaluation which could not be accomplished in this dissertation due to infrastructural limitations.

In this dissertation, we showed that all these techniques can be effectively combined to implement a transparent runtime program specializer. The current implementation has two shortcomings arising out of the limitations of our research infrastructure: (1) in the absence of runtime compilation support, we evaluate our specializer using emulation techniques, and (2) while we have proposed several techniques for detecting memory invariance violations, we have not implemented any technique and assumed the presence of a fine-grained memory protection scheme in our evaluation.

Despite the limitations of the specializer implemented in this dissertation, our evaluation shows that it provides good program speedups. The specializer was able to speedup an image processing

convolution kernel by 2X and was able to speedup a scheme interpreter by 3.6X.

10.2. Future Work

This work is the first step towards the goal of implementing specializers transparently within a dynamic optimizer. Our implementation has been geared towards simplicity, with the goal of investigating how powerful a simple specializer can be. Our technique is limited in a few ways, but we believe that more powerful run-time specializers can build upon our work, especially on the profile-driven dynamic analysis (that detects invariance and builds specialization scopes).

Throughout this dissertation, we have pointed out the restrictions of our model, the simplifying assumptions that have been made, the limitations of our techniques and left it to future work to address all these restrictions, assumptions, and limitations. In this section, we summarize some areas of future work.

10.2.1. Improved specialization model

The foremost area where future work can improve on the techniques proposed in this dissertation is in lifting the restrictions of our specialization model.

In Section 3.8, we presented the restrictions that has been imposed on the specialization model studied in this dissertation. One of the fundamental design choices in our technique was to restrict the specialization key to a single object reference variable. While this choice enabled us to design a simple scope building algorithm and allowed for efficient implementations of the lookup mechanisms, we lost specialization opportunities. For example, with two-element keys, we would be able to exploit specialization opportunities in `java.lang.String.equals` and other similar methods.

Secondly, by allowing scalars to be part of specialization keys, it is possible to improve performance further. For example, using scalars in specialization keys would enable chaining of specialized

versions of the interpreter loop shown in Figure 9.3, thus eliminating lookup overheads. We examine this further now.

Figure 9.3 shows a snippet of our interpreter benchmark. Let us suppose that it is interpreting the factorial assembler program shown in Figure 10.1. In this example, the bulk of the interpretation

```
1.        LI r31, 1
2.        LI r2, 0
3.   L0:  BEQ r1, r2, L1
4.        MUL r31, r1, r31
5.        SUBI r1, r1, 1
6.        GOTO L0
7.   L1:  RET
```

Figure 10.1. Factorial assembly program input to the interpreter shown in Figure 9.3

happens for the loop consisting of instructions 3–6. The specializer recognizes the "hotness" of the instruction objects representing these instructions and creates four specialized versions of the entire *switch* statement of Figure 9.3, named S_3, S_4, S_5, S_6, one each for instructions 3, 4, 5, and 6. The specialized interpreter performs a lookup in each iteration of the loop and dispatches to the appropriate specialized version. However, the specializer is not powerful enough to recognize that these specialized versions can be chained together (chaining is a transformation that would eliminate the three high-overhead lookups connecting the specialized versions: $S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_3$). Chaining is prevented in the specializer because the specialized interpreter body, shown in Figure 10.2, invokes a getarray operation between each specialized version.

```
for(; ;) {
   pc = μ(pc0, pc1);
   i = getarray(pgm, pc);
   lookup(i);
   ... specialized versions of the switch statement S...
   pc1 = φ(pc2, pc3, ...);
}
```

Figure 10.2. Specialized body of the interpreter loop shown in Figure 9.3

Clearly, the ideal scope in this example should include the entire iteration of the `for` loop and be keyed on the variable `pc`. The identified specialization scope, however, is smaller. It contains only the switch body and is keyed on the variable `i`. The reasons for this suboptimal decision is the initial design decision to restrict keys to object references, for efficiency's sake. Since `pc` is a scalar, it cannot serve as a key. This lost opportunity explains the somewhat disappointing performance on the interpreter benchmark (Table 9.6).

Extending the specializer to scalars is rather straightforward, at least in principle. In practice, some re-engineering of our specializer is necessary. For one, profiles would have to collected for scalar uses too, and it is not sufficient to collect value profiles at object accessing instructions. Secondly, the scope-building algorithm would have to be extended to handle scalar keys – using more sophisticated transfer functions for scalar instructions might be sufficient. Addressing this is left for future work.

10.2.2. Improved scope-building and specialization

In Chapter 5, we presented various simplifying heuristics using in building scopes. While developing this algorithm, the goal was efficient and fast detection of specialization scopes which led us to adopt a greedy strategy in building specialization scopes. However, our evaluation reveals that the algorithm studied in this dissertation is efficient and the implementation can be engineered better to further reduce runtime costs. In the light of this result, it might be worthwhile to explore other possible non-greedy scope-building strategies.

In Chapter 6, we presented a SCCP-based specialization algorithm. However, this algorithm only implements simple single-way unrolling. While this works well in many cases, in Section 6.3, we discussed how multi-way unrolling can enable better specialization for certain applications like the interpreter loop shown in Figure 9.3. This is one potential area for future work. Existing specializers like DyC [47] implement multi-way unrolling and the technique used there could potentially be

adapted to our specializer.

10.2.3. Systematic study of techniques that detect violations of memory invariance

In Chapter 8, we have proposed several techniques for detecting violations of memory invariance. However, due to infrastructure limitations, these techniques could not be implemented and studied. The proposed store-monitoring techniques need closer study since they appear quite promising with their potential for low overheads and integration with garbage collection write barriers.

BIBLIOGRAPHY

[1] Domain Specific Languages. http://compose.labri.fr/documentation/dsl/.

[2] ImageJ: Image Processing and Analysis in Java. http://rsb.info.nih.gov/ij/.

[3] Intel Open Source Dynamic Computing Research Platform. http://www.intel.com/research/mrl/orp/.

[4] Jikes RVM. http://oss.software.ibm.com/developerworks/oss/jikesrvm.

[5] Sprint - A framework for Developing DSLs. http://compose.labri.fr/prototypes/sprint/.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.

[7] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 85–96, 1997.

[8] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[9] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems*, 15(4):357–390, Nov. 1997.

[10] M. Arnold. *Online Profiling and Feedback-Directed Optimization of Java*. PhD thesis, Rutgers University, October 2002.

[11] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F.Sweeny. Adaptive Optimization in the Jalapeno JVM. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '00)*, Oct. 2000.

[12] M. Arnold, M. Hind, and B. Ryder. Online Feedback-Directed Optimization of Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '02)*, Nov. 2002.

[13] M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Proceedings of the 2001 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 168–179, June 2001.

[14] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, 21– May 1996.

[15] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Runtime Optimization System. In *Proceedings of the 2000 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 1–12, 2000.

[16] T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[17] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57. ACM Press, 1996.

[18] J. L. Bentley. *Writing Efficient Programs*. Prentice Hall, Englewood Cliffs, 1982.

[19] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings*

of the 15th ACM Symposium on Operating System Principles (SOSP-15) Copper Mountain, CO., pages 267–284, 1995.

[20] R. S. Bird. Tabulation Techniques for Recursive Programs. *ACM Computing Surveys*, 12(4):403–418, 1980.

[21] M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999.

[22] B. Calder, P. Feller, and A. Eustace. Value Profiling. *Journal of Instruction Level Parallelism*, March 1999.

[23] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu. Profile-guided Automatic Inline Expansion for C Programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[24] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software: Practice and Experience*, 21(12):1301–1321, Dec. 1991.

[25] W. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000.

[26] T. Chilimbi and M. Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *PLDI '02*, June 2002.

[27] J. Clifford Noel Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, 1995.

[28] N. H. Cohen. Eliminating Redundant Recursive Calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[29] D. Connors and W. Hwu. Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. In *32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 158–169, 1999.

[30] C. Consel, L. Hornof, François Noël, J. Noyé, and N. Volanschi. A Uniform Approach for Compile-time and Run-time Specialization. In *Partial Evaluation. International Seminar.*, pages 54–72, Dagstuhl Castle, Germany, 12-16 Feb. 1996. Springer-Verlag, Berlin, Germany.

[31] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. Tempo: Specializing systems applications and beyond. In *ACM CS*, page 30(3es):19 es, Sept. 1998.

[32] C. Consel and F. Noël. A General Approach for Run-Time Specialization and its Application to C. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 21–24 Jan. 1996.

[33] T. M. Conte, K. N. Menezes, and M. A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 36–45, Paris, France, dec 1996. ACM Press.

[34] Craig Zilles and Gurinder Sohi. A Programmable Co-processor for Profiling. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, Jan. 2001.

[35] D.Deaver, R.Gorton, and N.Rubin. Wiggins /Restone: An on-line Program specializer. In *In Proceedings of Hot Chips 11*, Aug. 1999.

[36] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th*

Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97), pages 292–302, Los Alamitos, Dec. 1–3 1997. IEEE Computer Society.

[37] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308 (Available from http://www.cs.wisc.edu/trs.html), University of Wisconsin-Madison, July 1996.

[38] E. Duesterwald and V. Bala. Software Profiling for Hot Path Prediction: Less is More. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Nov. 1–3 2000.

[39] C. Dulong. The IA-64 Architecture at Work. *Computer*, 31(7):24–32, July 1998.

[40] K. Ebcioğlu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In 24th *Annual International Symposium on Computer Architecture*, pages 26–37, 1997.

[41] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[42] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. mei W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 183–193, San Jose, California, 1994.

[43] W. G.Cochran. *Sampling Techniques*. John Wiley and Sons, 1977.

[44] L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[45] A. Gonzalez, J. Tubella, and C. Molina. Trace-Level Reuse. In *Proceedings of the International Conference on Parallel Processing*, September 1999.

[46] B. Grant. *Benefits and Costs of Staged Run-time Specialization*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 2001.

[47] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Technical Report TR-97-03-03, University of Washington, Department of Computer Science and Engineering, Mar. 1997.

[48] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. The UW Dynamic Compilation Project. Technical report, University of Washington, 1998.

[49] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. The benefits and costs of DyC's run-time optimizations. *ACM Transactions on Programming Languages and Systems*, 22(5):932–972, 2000.

[50] T. Heil. *Relational Profiling in Multithreaded Virtual Machines*. PhD thesis, University of Wisconsin, Madison, 2002.

[51] M. Hirzel and T. Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001.

[52] U. Hölzle. A Fast Write Barrier for Generational Garbage Collectors. In *OOPSLA'93 Conference Proceedings*, Washington, DC, 1993.

[53] U. Holzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Thesis CS-TR-94-1520, University of Stanford, Aug. 1994.

[54] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In P. America, editor, *ECOOP '91: European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, 1991.

[55] Jian Huang and David J. Lilja. Exploiting Basic Block Value Locality with Block Reuse. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 106–114, Orlando, Florida, Jan. 9–13, 1999. IEEE Computer Society TCCA.

[56] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[57] A. Klaiber. The technology behind Crusoe(tm) Processors, Jan. 2000.

[58] T. B. Knoblock and E. Ruf. Data Specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 215–225, Philadelphia, Pennsylvania, 21–24 May 1996.

[59] C. Krintz and B. Calder. Using Annotation to Reduce Dynamic Optimization Time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–167, 2001.

[60] C. J. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software-Practice and Experience*, 31(8):717–738, 2001.

[61] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *In Proceedings of the ACM SIGPLAN 96 Conference on Programming Language Design and Implementation Philadelphia, PA,USA*, pages 137–148, May 1996.

[62] K. M. Lepak, G. B. Bell, and M. H. Lipasti. Silent stores and store value locality. *IEEE Transactions on Computers*, 50(11):1174–1190, November 2001.

[63] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, USA, 1997.

[64] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 226–237, Paris, France, Dec. 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[65] R. Marlet, S. Thibault, and C. Consel. Mapping Software Architectures to Efficient Implementations via Partial Evaluation. In *Proceedings of the* 12th *IEEE International Conference on Automated Software Engineering (ASE 1997)*. IEEE, 1997.

[66] M.Burrows, U.Erlingson, S.T.A.Leung, M.T.Vandevoorde, C.A.Waldspurger, K.Walker, and W.E.Weihl. Efficient and Flexible Value Sampling. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 160–167, Nov. 1–3 2000.

[67] S. Meloan. The Java HotSpot (tm) Perfomance Engine: An In-Depth Look. Article on Sun's Java Developer Connection site, 1999.

[68] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[69] D. Michie. "Memo" Functions and Machine Learning. *Nature*, pages 19–22, Apr. 1968.

[70] M. Mock. *Automating Selective Dynamic Compilation.* PhD thesis, University of Washington, Department of Computer Science and Engineering, 2002.

[71] M. U. Mock, C. Chambers, and S. J. Eggers. Calpa: A Tool for Automating Selective Dynamic Compilation. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-33)*, pages 291–302, Dec. 2000.

[72] T. Mogensen. Partial Evaluation: Concepts and Applications. In *Partial Evaluation: Prac-*

tice and Theory, Diku 1998 International Summer School, Copenhagen, Denmark, July 1998, Lecture Notes in Computer Science, pages 1–19. Springer-Verlag, July 1998.

[73] R. Muth, S. Watterson, and S. Debray. Code Specialization Based on Value Profiles. In *Proceedings of the* 7th *International Static Analysis Symposium (SAS 2000)*, pages 340–359. Springer LNCS vol. 1824, June 2000.

[74] P. Norvig. JScheme: Scheme in Java. http://www.norvig.com/jscheme.html.

[75] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 109–121, New York, June 15–18 1997. ACM Press.

[76] E. Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, 1993.

[77] S. S. Sastry, R. Bodik, and J. E. Smith. Rapid Profiling via Stratified Sampling. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA-01)*, volume 29,3 of *Computer Architecture News*, pages 278–289, New York, June 2001. ACM Press.

[78] Satish Narayanasamy and Timothy Sherwood and Suleyman Sair and Brad Calder and George Varghese. Catching Accurate Profiles in Hardware . In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, Feb. 2003.

[79] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 248–258, Los Alamitos, Dec. 1–3 1997. IEEE Computer Society.

[80] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards Automatic Specialization of Java Programs. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 367–390, Lisbon, Portugal, June 1999. Springer-Verlag.

[81] U. P. Schultz, J. L. Lawall, and C. Consel. Specialization Patterns. In *Proceedings of the* 15th *IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 197–206, Grenoble, France, Sept. 2000. IEEE.

[82] T. Sherwood and B. Calder. Time Varying Behavior of Programs. TechReport CS99-630, University of California-San Diego, Aug. 1999.

[83] J. E. Smith, T. Heil, S. Sastry, and T. Bezenek. Achieving High Performance via Co-Designed Virtual Machines. In *International Workshop on Innovative Architecture*, Oct. 1999.

[84] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, pages 194–205, June2–4 1997.

[85] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.

[86] S. Thibault, L. Bercot, C. Consel, R. Marlet, G. muller, and J. Lawall. Experiments in Program Compilation by Interpreter Specialization, Dec. 1998.

[87] S. Thibault, J. Marant, and G. Muller. Adapting distributed applications using extensible networks. In *ICDCS '99, Austin Texas*, pages 234–243, June 1999.

[88] Timothy Heil and James E Smith. Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-33)*, Dec. 2000.

[89] M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.

[90] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct. 1–3 2002.

[91] C. Young and M. D. Smith. Better Global Scheduling Using Path Profiles. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 115–126, Los Alamitos, Nov. 30–Dec. 2 1998. IEEE Computer Society.

[92] B. Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado at Boulder, (Revised) August 1992.