No dissertation is truly the work of a single individual, although by tradition only a single author is listed. I owe a deep debt of gratitude to the many people who supported me throughout this work, both personally and professionally.

My advisor, Jim Smith, has excelled as a mentor. His gentle and direct guidance continuously encourage improvement. His broad experience has enabled me to appreciate and understand the differences between academia and industry. When combined with his keen insight, his qualities form a model to be emulated throughout my career.

Professor Ras Bodik generously gave his expertise in those areas where I had little experience. His valuable counsel with respect to compiler optimization has improved the quality of this dissertation, particularly the work on object-oriented co-location.

Several fellow graduate students helped me take advantage of graduate school life. Informal technical discussions with Eric Rotenberg and Quinn Jacobson were one of the great joys of my early stay at Madison. Eric and Quinn taught me to take technical risks, and think in the large ...to look past details for the profound. Yanos Sazeides was a great encouragement during my initial research at Madison. Yanos helped me adjust to graduate school life, and understand the value and process of research in computer architecture.

Working with Subramanya Sastry on the Strata VM has been an entirely enjoyable experience. Subbu is a talented software engineer and researcher, and I have reaped the benefits of this talent. This research is built on the infrastructure of which he is the primary author. I was also fortunate to interact early with Ravi Rajwar on SimpleMP. The ready and continuous support he provided was a great aid, as was his willingness to discuss all manner of related issues, particularly memory consistency.

Interactions with many other students and faculty have improved my stay at Madison. Professor Saluja has always had an encouraging word, and a welcome smile. Todd Bezenek and Min Zhong also contributed to the developement of the Strata VM. Manoj Plakal was our first guinea pig (outside our own research lab), and has patiently supplied numerous improvements and bug fixes. Craig Zilles, with whom I had the joy of touring Israel following MICRO-32, and Milo Martin kindly discussed both career and research with me, giving valuable advice.

Finally, I have been blessed with a loving and caring family. My wife, Britt, has been a constant encouragement to me. Her patience has been unfailing through the months of hard work and long hours. The evident joy and pride shown be my parents, George and Jeanette, have helped me to maintain a sense of value in my work. Britt's mother, Bonnie Johnson, provided many good meals and has always helped me to relax. Maintaining a balance between work and life has been made easier by the caring friends and family, too many to mention here, who have been apart of my life here in Madison. Thank you.

This work was supported by NSF grants CCR-9900610 and EIA-0071924, Intel, IBM and Sun Microsystems. I would like to particularly thank Mario Wolczko, my supervisor during my internship at Sun Microsystems Labs, and Hal Kossman, my supervisor during my internship at IBM, Rochester.

> But beyond this, my son, be warned: the writing of many books is endless, and excessive devotion to books is wearying to the body. The conclusion, when all has been heard, is: fear God and keep His commandments, because this applies to every person. (Ecclesiastes 12:12,13)

ABSTRACT

Virtual machines (VMs) have been used to accelerate object-oriented applications and provide object-oriented features.

Virtual machines can supply ample amounts of *thread-level parallelism* (TLP) in the form of *service threads*. This is a natural fit for future microprocessors, which are likely to have abundant support for TLP. Service threads perform tasks such as garbage collection, profile analysis, and dynamic optimization in parallel with the main application thread(s).

Most service threads must monitor the application's behavior in order to be effective. The *relational profiling architecture* (RPA) is designed to enable efficient interaction between the application and the service threads within a multithreaded VM. To this end, the RPA provides the following features:

1. The RPA collects information from selected instructions under VM control. The RPA can be used for *instrumentation*, in which case the information is guaranteed to be collected, or it can be used for traditional *profiling*, in which case random sampling is used.

2. The RPA collects the information needed, and only the information needed, for these instructions.

3. The RPA examines this information for patterns, and discards information that is not of interest.

4. The RPA efficiently passes useful information to service threads through shared memory without interrupting the application. Service threads receive a stream of standard-format records, which they read and consume using normal loads and stores.

These features are provided by a low cost implementation based on two main components. The *profile control table* selects instructions for profiling during instruction dispatch, providing features 1 and 2. The query engine examines the information for patterns, and sends this information to service threads, providing features 3 and 4.

Two applications of the RPA illustrate its value.

1. A concurrent garbage collector uses the RPA to monitor application writes to objects. Service threads consume this information and perform the *write barrier* needed for correctness by the garbage collector.

   The VM identifies store instructions that need to be examined using a per-instruction profile tag provided by the RPA. The RPA essentially instruments these instructions, guaranteeing that need information is collected—a necessity since the information is required for correctness. The RPA filters out writes of null references, reducing the number of records processed by the service threads and simplifying the code they execute. The resulting garbage collector has an average overhead less than 0.4%.

2. Speculative object co-location improves spatial locality and reduces pointer chasing by placing related objects in adjacent memory. Speculative object co-location is driven by profiles of object allocations and accesses.

   Profiling overhead is reduced through *adaptive sampling*. The sampling rate for each instruction is adjusted independently. Rarely executed instructions are frequently sampled; frequently executed instructions are rarely sampled. The architectural features of the RPA allow the object co-location profiling task to be combined with the garbage collection profiling task, even though the two task have different requirements for the same instructions. The ability to compose multiple profiling tasks is likely to be an important characteristic in a large VM development project.

Contents

1 Introduction **1**

1.1 Virtual Machines 4

1.2 Service Threads and the Virtual Machine 5

1.3 The Relational Profiling Architecture and Service Threads 6

1.4 Evaluating the RPA 7

1.4.1 Concurrent Garbage Collection 9

1.4.2 Speculative Object Co-Location 11

1.5 Thesis and Contributions 12

1.6 Dissertation Organization 12

2 Related Work **14**

2.1 Previous Co-Designed Virtual Machines 14

2.1.1 VMs for High-Level V-ISAs 15

2.1.2 VMs for Low-Level V-ISAs 18

2.2 Mechanisms for Thread-Level Parallelism 19

2.3 Work Related to Service Threads 23

3 The Relational Profiling Architecture **27**

3.1 The Relational Profiling Model 28

3.2 The RPA Assembly Language 31

3.2.1 Consuming Messages Efficiently 36

3.2.2 Examples 39

3.3 Summary 41

4 RPA Implementation and Performance **42**

4.1 Hardware Structures . . . 42

4.1.1 The Profile Control Table (PCT) . . . 42

4.1.2 The Query Engine . . . 43

4.2 RPA Implementation and Cost . . . 44

4.2.1 The Profile Control Table . . . 45

4.2.2 The Query Engine . . . 45

4.2.3 The Profile Networks . . . 46

4.2.4 Profiling Overhead . . . 48

4.3 System Software Issues . . . 49

4.3.1 Dead-Locks . . . 49

4.3.2 Virtual Memory . . . 52

4.3.3 Context Switching . . . 52

4.4 Infrastructure and Methodology . . . 53

4.4.1 The Strata VM . . . 53

4.4.2 SimpleMP Simulator . . . 60

4.4.3 Baseline Microarchitecture . . . 60

4.4.4 Benchmarks . . . 63

4.4.5 Benchmark Measurement Intervals . . . 64

4.4.6 Simulated Queries . . . 64

4.5 Performance Analysis of the RPA . . . 66

4.5.1 Resource Usage . . . 66

4.5.2 Performance Overhead . . . 70

4.5.3 Profiling Rate . . . 73

4.5.4 Bus Bandwidth . . . 73

4.5.5 Profile Network Latency Tolerance 77

4.6 Related Work 79

4.7 Summary 81

5 Concurrent Garbage Collection Using the RPA 83

5.1 Concurrent Garbage Collection 84

5.1.1 Concurrent Mark-Sweep GC 85

5.1.2 Barriers 85

5.1.3 Previous Hardware Support 88

5.2 The Algorithm 92

5.2.1 GC State 92

5.2.2 Concurrent Modifications of the Heap 93

5.2.3 Heap Organization 93

5.2.4 Concurrent Algorithm 96

5.2.5 Correctness 98

5.3 Generational Collection 99

5.3.1 Generational Barriers 100

5.3.2 Using the RPA 101

5.3.3 Removing Objects from the Remembered Set 101

5.3.4 Generational Heap Organization 103

5.3.5 Algorithmic Extensions 103

5.4 Results 105

5.4.1 GC Performance Overhead 106

5.4.2 GC Phases 108

5.4.3 Filtering Null-Reference Writes 111

5.5 Summary 111

6 Speculative Object Co-Location 114

6.1 Overview 116

6.2 Speculative Object Co-Location Transformations 119

6.2.1 Transforming Object Layout 119

6.2.2 Transforming Allocation Code 121

6.2.3 Transforming *getfields* 122

6.2.4 Required Invariants 126

6.2.5 Integration with Garbage Collection 126

6.3 Ideal Profiler 126

6.3.1 Profiled Information 127

6.3.2 Co-Location Topologies 127

6.3.3 Recognizing Co-location Topologies 129

6.4 Reaching Definitions Analysis 130

6.5 Field Selection 132

6.5.1 Performance Model 134

6.5.2 Initial Filter 134

6.5.3 Check Reaching-Definitions Analysis 135

6.5.4 Check Array Sizes 136

6.5.5 Check for Cyclic Co-Location 136

6.5.6 Topology Analysis 136

6.5.7 Filter Insignificant Fields 139

6.5.8 Check for Reverse Allocation 139

6.6 Simulation Methodology 140

6.6.1 Application Execution 141
6.6.2 Measurement Intervals 142
6.7 Results 143
6.7.1 Observed Code Idioms 145
6.7.2 Co-Located Fields 149
6.7.3 Instruction Stream 150
6.7.4 Performance 155
6.7.5 Data-Cache Performance 155
6.7.6 Instruction-Cache Effects 159
6.7.7 Branch Prediction 163
6.8 Co-Location Profiling Using the RPA 165
6.8.1 The RPA Queries 166
6.8.2 Adaptive Sampling 168
6.8.3 Topology Recognition 169
6.9 Simulation Methodology with RPA-Based Profiling 170
6.9.1 Collecting Profiles 170
6.9.2 Profile Overhead 171
6.10 Results Using the RPA-Based Profiler 171
6.10.1 Fields Co-Located 172
6.10.2 Performance of Co-Location Using the RPA-Based Profiler 173
6.10.3 Profiling Overhead of the RPA-Based Profiler 173
6.10.4 Profile Algorithm Execution Overhead 176
6.11 Related Work 177
6.11.1 Object Inlining 177
6.11.2 Object Co-Location 179

6.11.3 Data Restructuring 181

6.12 Summary 183

7 Summary and Conclusions 185

7.1 Architectural Features 185

7.1.1 Intelligent Instruction Selection 186

7.1.2 Instruction Instrumentation 186

7.1.3 Boolean Pattern Matching 186

7.1.4 Shared-Memory Message Passing 186

7.2 Implementation 187

7.3 Applications of the RPA 188

7.3.1 Concurrent Garbage Collection 188

7.3.2 Speculative Object Co-location 189

7.4 Architectural Benefits 189

7.4.1 Message Passing 190

7.4.2 Per-Instruction Profile Tag 190

7.4.3 Instruction Instrumentation 191

7.4.4 Boolean Pattern Matching 191

7.4.5 Composing Profiling Tasks 191

7.5 Experience with Service Threads 192

7.6 Further Research 193

7.6.1 Further Architectural Extensions 193

7.6.2 Further Profiling Applications 194

7.6.3 Automatic Composition of Queries 194

List of Figures

1.1 A virtual machine. 3

1.2 A multithreaded Java virtual machine. 6

1.3 The system-on-a-chip used to evaluate the RPA. 8

1.4 Object co-location improves spatial locality and reduces pointer-chasing. . . . 10

2.1 Support for thread-level parallelism. 20

3.1 The relational profiling model views instructions and events as a table. 28

3.2 The profiling process with the RPA. 30

3.3 Example RPA assembly language queries. 31

3.4 The RPA sends profile records to message queues in pools. 37

3.5 Service threads efficiently consume records four at a time. 38

4.1 An implementation of the RPA contains the profile control table (PCT) and the query engine. 42

4.2 Query instructions contain up to two comparisons, a branch and an action. . 44

4.3 The query engine is a simple four stage pipeline. 46

4.4 A common dead-lock situation with profiling. 49

4.5 Infrastructure used for this research. 54

4.6 Strata compiler optimization phases. 56

4.7 System-on-a-chip design. 61

4.8 Histogram of profile buffers occupied. 67

4.9 Histogram of in-flight profiled instructions. 69

4.10 Dispatch stalls vs. available profile networks. 71

4.11 Profiling overhead vs. available profile networks. 72

4.12 Profiling overhead vs. branch sampling rate. 74

4.13 MOESI protocol transactions associated with RPA messages. 76

4.14 Latency tolerance of the profile network. 78

5.1 Example concurrent reference mutation. 86

5.2 Write barrier for concurrent modifications. 93

5.3 The concurrent GC algorithm. 95

5.4 Removing objects from the remembered set creates a subtle synchronization hazard. 102

5.5 The generational GC algorithm. 104

5.6 Time line for the first GC in the Strata benchmark. 110

6.1 Object co-location improves spatial locality and reduces pointer-chasing. . . . 115

6.2 Speculative object co-location involves four points in the application code. . . 117

6.3 The speculative object co-location transformation is driven by three analysis steps. 118

6.4 Co-Location areas are added to container objects to make room for future co-located objects. 119

6.5 A global pointer is used to find available co-location areas. 120

6.6 Pseudo-assembly representing the optimization of a *getfield*. 122

6.7 Transformation and optimization of a co-locatable *getfield*. 124

6.8 Optimizing speculative sources requires live-range analysis. 125

6.9 Speculative object co-location uses profiling to distinguish different relationships or topologies between objects. 128

6.10 Decision tree used to recognize co-location topologies. 131

6.11 Reaching-definitions analysis locates allocation sites from *putfields*. 132

6.12 Field selection process. 133

6.13 Example topological data. 137

6.14 Allocating the co-located object first is problematic for the current scheme. . 139

6.15 If the co-located object is allocated first, the container class must be known exactly. 141

6.16 Construction of a *Face* object in *raytrace* creates a co-locatable object. 146

6.17 Array sizes are often available when the container object is allocated. 147

6.18 Enumerators are common, but difficult to co-locate. 148

6.19 Additional instructions due to dynamic checking. 153

6.20 Correlation between speedup and data-cache miss rate. 158

6.21 Instruction cache miss rates. 159

6.22 Performance lost due to instruction cache misses. 160

6.23 Instruction cache miss rates, vs. size for *xp*. 161

6.24 Performance loss due to I-cache misses for *xp*. 161

6.25 Performance variation due to object file link order for *xp*. 162

6.26 RPA queries used during object co-location profiling. 167

6.27 Profiling overhead without adaptive sampling 175

6.28 Profiling overhead with adaptive sampling 175

List of Tables

1.1 Object-oriented language features 2

3.1 Instruction profiling classes. 32

3.2 Information the RPA can collect. 34

3.3 Query clause comparison types. 34

4.1 Message engine PCRs. 44

4.2 RPA component costs. 45

4.3 Baseline processor model parameters. 62

4.4 Benchmark description. 63

4.5 Measurement interval heartbeats (thousands). 65

4.6 Measurement interval sizes (millions). 65

4.7 Estimated sampling rates yielding 2% profiling overhead. 75

4.8 Estimated bus bandwidth used by RPA messaging. 76

5.1 GC performance characteristics. 106

5.2 Mark phase execution characteristics. (thousands of cycles) 109

5.3 Sweep phase execution characteristics. (thousands of cycles) 109

5.4 Write-barrier work eliminated by the RPA. 112

6.1 Measurement intervals (thousands of heart beats). 144

6.2 Instructions in measured intervals (millions). 144

6.3 Cycles in measured intervals (millions). 145

6.4 Number and type of fields co-located, based on ideal profiling. 150

6.5 Co-Location instruction counts. 151

6.6 Performance. 156

6.7 Data-cache effects. 157

6.8 Branch prediction effects. 164

6.9 Fields co-located with RPA-based profiling, compared to ideal profiling 172

6.10 Performance of co-location, driven by RPA-based profiling. 174

6.11 *getfield* profiling rate and processing rate. 176

7.1 RPA component costs. 187

CHAPTER 1

INTRODUCTION

Object-oriented programming has become the preferred paradigm for software development. This popularity stems from powerful features that support productive software development techniques. Although these features are valuable, implementing them efficiently has been challenging.

Virtual machines (VMs) address this challenge by using mechanisms such as dynamic compilation and optimization. This dissertation proposes instruction set architecture (ISA) extensions to further increase the performance of object-oriented programs executing under a VM. These extensions are embodied in the *relational profiling architecture* (RPA). The RPA assists in the collection and communication of information within a multithreaded VM. It does so by selectively collecting information from instructions and communicating this information to VM *service threads* without interrupting the application. This enables the service threads to perform tasks that improve application performance and support object-oriented features.

Table 1.1 lists some features supported by the object-oriented language, Java [11]. Java was chosen as the vehicle for this research because it is a popular object-oriented language, supports powerful language features, and it is intended for execution via virtual machines.

As shown in Table 1.1, object-oriented languages create and manipulate objects. Objects wrap code and data into a single unit. Object-oriented languages assure that objects are only used the way they are intended. This allows software developers to rigorously enforce important invariants that make reasoning about applications easier.

Garbage collection (GC) automatically reclaims memory no longer used by the application. GC simplifies program logic, and eliminates dangling pointers and memory leaks. Using virtual function calls, different types of objects can implement the same interface in

Table 1.1. Object-oriented language features

Language Feature	*Efficient Implementation Techniques*
Memory allocated in objects	Data-restructuring, object inlining, object co-location
Type-safety and access checks	Advanced compilation techniques
Garbage collection	Concurrency, hardware support
Virtual function calls	Dynamic compilation
Dynamic linking	Dynamic compilation

different ways, as appropriate for each type. Virtual function calls improve modularity and code-reuse because a single piece of code can manipulate many types of objects that all support the same interface. Dynamic linking finds and links application code at run-time, rather than statically at compile time. Dynamic linking aids in the extension and maintenance of library code.

These features are helpful. But, they often constrain the performance of object-oriented applications. In some cases, clever compilation techniques can remove object-oriented overheads. For instance, techniques are known for removing array-bounds checks [19] and null-reference checks [5].

Other issues are more difficult to address at compile time. For instance, garbage collection is a complex process that uses 20% or more of the CPU cycles [10, 100, 145, 71, 83]. Also, objects are typically allocated at arbitrary addresses that are hidden from the programmer by object-oriented semantics. Objects can only be found by following references from one object to another. This reduces spatial locality and increases pointer-chasing, compared to languages like C where structures can be directly embedded within one another.

The modularity that object-oriented languages support inhibits many common compiler optimizations. Object-oriented techniques tend to produce many small functions. Since functions are the typical scope for most compiler optimizations, small functions leave little room for optimization.

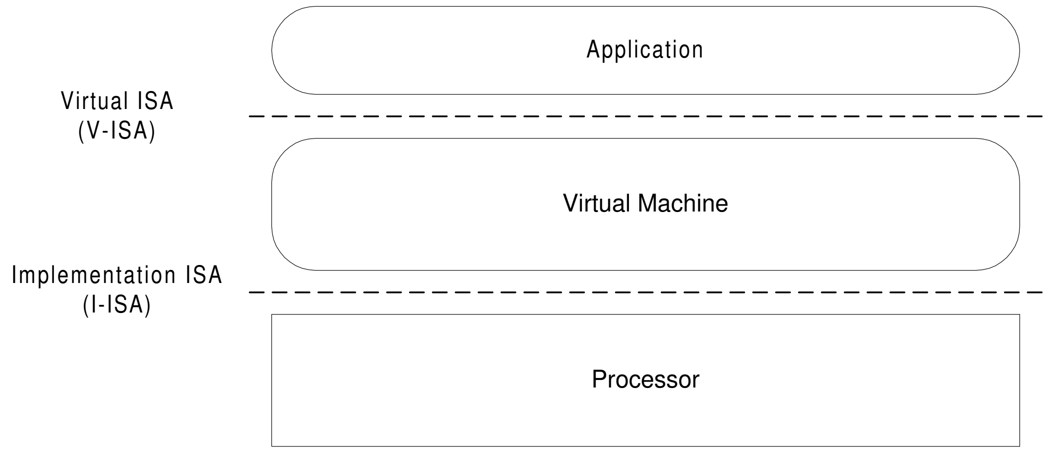


Figure 1.1. A virtual machine.

Interprocedural optimization techniques [66, 80] and function inlining [13] widen the scope of optimization for small functions. However, these techniques are hampered by virtual function calls and dynamic linking. Without run-time information regarding what call targets are common, interprocedural analysis becomes conservative and function inlining becomes less effective. Dynamically loaded functions, small or large, are not known at compile time, making analysis and inlining infeasible.

Virtual machines provide a promising framework for efficiently implementing object-oriented features. Since VMs delay some portion of compilation until run-time, a whole host of information is available to the VM's dynamic compiler that is not available to a static compiler. For instance, dynamically loaded code is available for inlining and interprocedural analysis. Profile information that illuminates program behavior is also available. Call-edge profiles indicate the likely target of virtual function calls so that virtual calls can be optimized and/or inlined [68, 92]. Profiles of object accesses drive the optimization of object layout and arrangement (See Section 6.11).

1.1. Virtual Machines

VMs execute applications presented in one architecture on hardware supporting another. See Figure 1.1. The application is coded in the *virtual instruction set architecture* (V-ISA). The VM uses a combination of interpretation and compilation to execute this application on hardware supporting a different *implementation ISA* (I-ISA).

VMs execute a wide variety of V-ISAs. For instance, the Transmeta Crusoe processor executes the IA-32 ISA on a VLIW processor [62, 78]. Emulating a complete instruction set that was designed for direct execution is challenging [55]. However, *high-level V-ISAs*, such as Java bytecodes [87] and the DotNET common language runtime [63] are designed with VMs in mind. These V-ISAs generally cannot be executed without a VM. High-level V-ISAs make the design of the VM considerably more flexible because they communicate high-level program structure to the VM and hide implementation details.

VMs decouple program representation from hardware implementation. *Co-designed virtual machines* take advantage of this abstraction to modify the I-ISA and CPU microarchitecture [64, 78, 114, 91, 55]. Co-designed VM hardware and software are developed together for improved performance, power efficiency, or both.

The I-ISA is flexible across generations because the V-ISA provides the necessary binary compatibility. By softening the requirement of backward compatibility for the I-ISA, co-designed VMs accelerate and simplify the absorption of microarchitectural and technological advances. This approach is used by the A/S 400 [116], which switched from a co-designed CISC architecture to the PowerPC architecture [147, 35].

The goal of this dissertation is not to entirely redefine the architecture of the CPU, as with the Crusoe processor [62, 78]. Instead, we seek to support desirable software primitives for profiling with minimal disruption to the I-ISA. The RPA extends the I-ISA in two ways.

First extra instructions are created for configuring the profile hardware. Second, a two-bit profile tag is added to each instruction to allow the VM to select particular instructions for profiling.

We place the profile tag directly inside the instruction because this is a simple solution. Although the co-designed VM paradigm supports changes to instruction formats, the additional two bits may still be prohibitive. If so, the profile bits can be kept in another location, possibly determined through address translation. Another alternative is to use special instructions to specify the profile tags for the following dynamic instructions. This technique has been evaluated for guarded execution [102]. An advantage of these techniques is that only software that uses the tag bits pays the overhead for them.

1.2. Service Threads and the Virtual Machine

Increasing transistor counts, decreasing cycle times, and increasing wire delays indicate that future technologies will likely have ample support for multithreaded execution. Recent processor designs support this trend. For example, the Pentium 4 incorporates simultaneous multithreading (SMT)[89], and the Power4 is a chip multi-processor [46]. Other examples are discussed in Section 2.2.

In contrast to *instruction-level parallelism* (ILP), *thread-level parallelism* (TLP) is relatively easy to support in hardware. ILP-oriented superscalar processor designs now consume enormous hardware resources, transistors, area and power, to reap even incremental gains in instructions per cycle. Studies of SMT [128, 140, 141, 60] and CMP [16, 99], show that processors designed to exploit TLP, in addition to ILP, can obtain much greater throughput with the same resources.

Of course, this requires multiple threads of execution, and VMs provide a promising way of harnessing these multiple threads. Superscalar designs focus on ILP primarily because

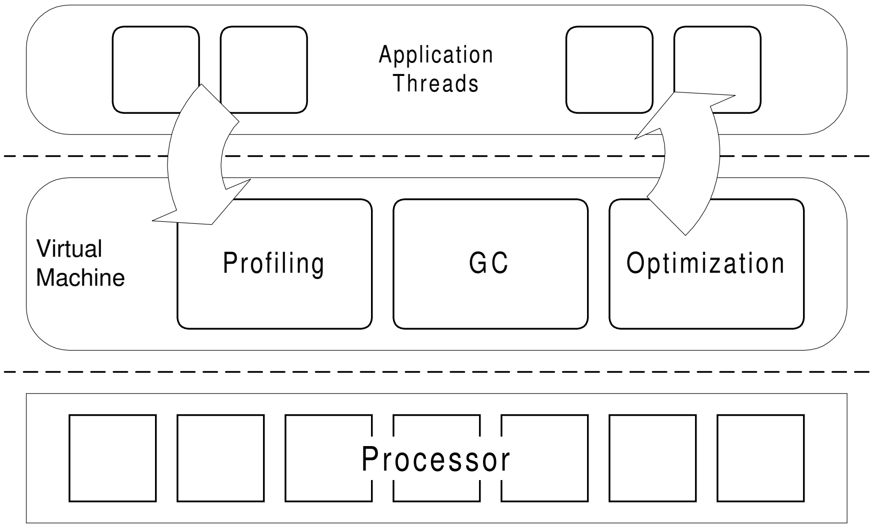


Figure 1.2. A multithreaded Java virtual machine.

multiple threads have only rarely been available in a general-purpose environment. High performance VMs have the potential to change this.

A co-designed VM can perform many tasks concurrently with the execution of the main program threads. These include garbage collection, profile collection and analysis, and dynamic optimization. Furthermore, each of these tasks can be parallelized itself. We refer to threads performing these auxiliary tasks as *service threads*. The service threads and main program thread(s) collectively lead to higher performance using relatively simple TLP. These techniques are already in use by two VMs for Java: Jikes and HotSpot [92, 12]. Section 2.3 discusses other examples of the service thread paradigm.

1.3. The Relational Profiling Architecture and Service Threads

Figure 1.2 provides a high-level view of a multithreaded co-designed VM. An application is executed by the VM on hardware with abundant support for multiple threads. While the application thread(s) execute, VM service threads concurrently optimize the application, collect and process profile information, and perform garbage collection.

Efficient interaction between application thread(s) and service threads is crucial for multithreaded co-designed VMs. To be effective, service threads must respond efficiently and correctly to the application's requirements. The RPA, is designed to enable such efficient interaction.

The RPA provides an expressive, uniform interface for collecting information from application threads, and communicating information to service threads. First, the RPA selects particular instructions based on the op-code and the profile tag. Second, the RPA collects specific information from the selected instructions. When the RPA is used for instrumentation, the information is guaranteed to be collected from each dynamic instance of selected instructions. The RPA can also perform random sampling, for use as a traditional profiler. Third, the RPA evaluates collected information with arbitrary Boolean expressions. Using this capability, the RPA looks for information with desired traits, and discards unneeded information. Fourth, the RPA communicates collected information to service threads via shared-memory queues. Information is written automatically by the RPA without interrupting the application thread. Service threads efficiently read this information using normal loads and stores, and then perform further processing.

To use the RPA, a VM developer describes a profiling task in *queries* using the RPA assembly language. The queries are assembled to form an RPA binary. When the VM starts, the binary is down-loaded to the RPA hardware. Once enabled, the RPA begins collecting the information specified by the query.

1.4. Evaluating the RPA

The RPA is evaluated along three dimensions – expressiveness, implementation and performance. This evaluation begins with a detailed implementation of the RPA. The implementation provides a basis for evaluating the hardware costs associated with the RPA. From

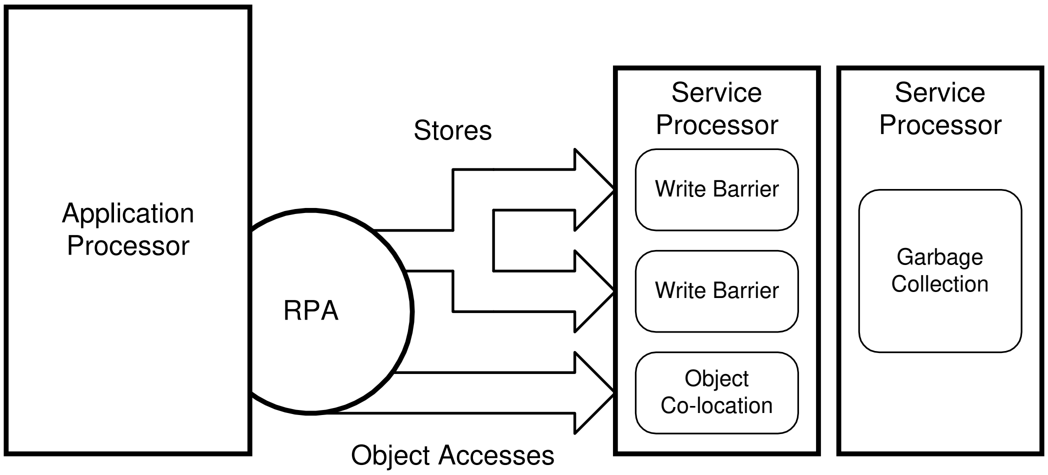


Figure 1.3. The system-on-a-chip used to evaluate the RPA.

this implementation, a cycle-level simulation model is developed to measure the overhead of using the RPA. Expressiveness and performance are evaluated through detailed development of two of many possible profiling applications.

The RPA is efficiently implemented with two main components. The *profile control table* (PCT) selects instructions for profiling early in the pipeline, and allocates the hardware resources needed for profiling. When profiled instructions retire, the *query engine* checks the collected information for patterns, and passes selected profile records to service threads through shared memory. The query engine is a simple four-stage pipeline capable of performing Boolean pattern matching. Its implementation is simplified by its minimal architected state and its multithreaded design.

The benefits of the RPA are explored using detailed cycle-level timing simulations of the three-processor system-on-a-chip shown in Figure 1.3. All processors have the same I-ISA; the microarchitectures are different. The application processor is an out-of-order core designed to provide high-performance for high-priority application threads. The service processors are small in-order scalar pipelines designed to provide high throughput for many service threads. Each service processor can execute three threads using fine-grained multithreading.

A Java application runs on the application processor under the control of a co-designed VM, in this case Strata, a research VM for Java. While the application executes, the RPA gathers needed information and sends it to service threads executing on the service processors. Two write-barrier service threads process profile information on behalf of the garbage collector, as described in the next subsection. A third service thread consumes profiles for speculative object co-location, as described in Subsection 1.4.2.

There are many ways of supporting multiple threads of execution, as Section 2.2 describes. In this dissertation heterogeneous CMPs, like the one shown in Figure 1.3, are simulated to evaluate the RPA. A comprehensive comparison of alternatives is beyond the scope of this work.

1.4.1. Concurrent Garbage Collection

Concurrent garbage collection provides an excellent opportunity to demonstrate RPA-driven service threads. Garbage collectors find and reclaim memory no longer needed by an application. Concurrent garbage collectors do this while the application is running. Because the application may modify objects being concurrently examined by the collector, concurrent collectors use a *write barrier* to observe all application modifications to the heap.

Write barriers have been implemented with inlined write-barrier code, resulting in high overhead, and with specialized hardware, resulting in loss of generality. The RPA efficiently performs this function with the flexibility of inlined write barriers and the performance of dedicated hardware support.

To perform the write barrier, the VM uses the profile tag to identify store instructions that modify objects. The RPA profiles these store instructions, and communicates the mod-


```
Y = LOAD [X + field_offset]
...
```

a) Without co-location.

```
Y = X + co_loc_offset
Y' = LOAD [X + field_offset]
if ( Y ≠ Y' ) Y = Y'
...
```

b) With co-location.

Figure 1.4. Object co-location improves spatial locality and reduces pointer-chasing.

ified address and the written value to service threads via the shared-memory communication interface. The service threads perform the write barrier needed by the collector for correctness.

This application demonstrates several of the RPA's features. For correctness, GC must examine *every* relevant store. This form of instrumentation is a key strength of the RPA The RPA's query engine performs a required null-value check, reducing the workload on the service threads. The share-memory communication mechanism eliminates expensive interrupts, and allows write barrier code to be executed concurrently. Finally, the behavior of the write barrier changes slightly, depending on the phase of collection. Changing the behavior of the barrier is easy with the RPA. The service threads simply check a global variable to determine what type of barrier to execute. Such changes would be difficult with inlined barrier code. This result is a concurrent garbage collector with an average run-time overhead of 0.4%.

1.4.2. Speculative Object Co-Location

Speculative object co-location allocates multiple objects together to improve cache locality and reduce pointer-chasing overheads. As shown in Figure 1.4, when one object, called the *container object*, references another, the *co-located object*, in a stable manner, the co-located object can be allocated within the container object. The offset of the co-located object within the container object is fixed and known to the VM, allowing *getfields* of the reference to be speculatively converted into add instructions.

Speculative object co-location demonstrates the RPA's capabilities as a profiler. To support this optimization, the RPA collects information on object allocations, array sizes, reference reads and reference writes.

Adaptive sampling is a powerful capability demonstrated by this application. Service threads dynamically adapt the profiling rate to the execution frequency of the instructions. The sampling rate is controlled on a per-instruction basis using the profile tags provided by the RPA. For rarely executed instructions high sampling rates provide the accurate estimates that are essential to object co-location. For frequently executed instructions low sampling rates reduce the profiling overhead. The resulting profiling overhead is less than 3%, provided the program executed long enough for the profiler to adapt.

For this application, the RPA simultaneously performs five different profiling tasks, three for co-location profiles, one for concurrent garbage collection, and one for edge profiling. Two of these tasks have different profiling requirements for the same instructions. The object co-location profiler has to collect PCs of store instructions with an adaptive random sampling rate. The concurrent GC algorithm has to collect the input values for the same stores, but without random sampling. The RPA performs both tasks simultaneously, providing evidence of its flexibility. Multiple profiling tasks for different purposes can be combined, even when the tasks have different profiling requirements for the same instructions.

1.5. Thesis and Contributions

This dissertation defends the conclusion that *the relational profiling architecture efficiently supports a wide range of profiling and instrumentation with intelligent instruction selection, shared memory communication and a low-cost implementation.* The following contributions support this thesis.

1. Development of the relational profiling architecture.

   A detailed description of the RPA and the reasoning behind it provides a basis for future development of derivative mechanisms. Descriptions of how the RPA performs several common tasks provide initial support for its generality.

2. Development and exploration of an implementation of the relational profiling architecture.

   Demonstration and simulation of one reasonable implementation of the RPA shows that the architected features are feasible, increasing confidence in the research. Simulation studies show that profiling overheads can be made very low, often less than 1%, given sufficient and reasonable resources.

3. Development of two applications of the RPA.

   Complete development of two RPA-based optimizations, concurrent garbage collection and speculative object co-location, provides quantitative evidence of the RPA's usefulness. Each of these optimizations is a significant contribution in themselves.

1.6. Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 describes prior work on co-designed virtual machines, hardware support for multithreading, and service threads. Work related to other topics is discussed within the appropriate chapters. Previous work on hardware assisted profiling is discussed in Section 4.6. Previous work on concurrent garbage collection can be found in Section 5.1. Section 6.11 covers previous work related to speculative object co-location.

Chapter 3 describes and illustrates the RPA using some simple profiling tasks. Chapter 4 describes an efficient implementation of this architecture, using timing simulations to investigate the tradeoff between profiling overhead and hardware resources. Chapter 5 describes the concurrent GC algorithm based on the RPA. Chapter 6 discusses speculative object co-location. Chapter 7 summarizes and concludes this dissertation.

CHAPTER 2

RELATED WORK

This dissertation combines several topics—virtual machines, hardware-assisted profiling, multiprocessing and multithreading, service threads, garbage collection, and object co-location. Where applicable, previous work is discussed in the chapter related to that topic. Previous work on hardware assisted profiling is discussed in Section 4.6. Previous work on concurrent garbage collection can be found in Section 5.1. Section 6.11 covers previous work related to object co-location.

Work related to overarching topics is covered in this chapter. Section 2.1 covers prior work on co-designed virtual machines. Mechanisms for supporting multiple threads are discussed in Section 2.2. Previous work related to the service thread paradigm is described in Section 2.3.

2.1. Previous Co-Designed Virtual Machines

This dissertation builds on a long history of VM research. Previous VMs have been studied and constructed for portability, isolation and security, replication, and performance.

VMs emulate one environment on top of another. This entails translation of O/S and standard library calls and/or instructions— either user-level, system-level or both. Hence, VMs can be characterized by the platform upon which the VM must run (i.e. what is below the VM software) and the environment that the VM must support (i.e. what is above the VM software).

For instance, VMWare [118] executes one or more IA-32 operating systems on top of a possibly different IA-32 O/S. The ISA remains unchanged, but the O/S must be emulated. In contrast, FX!32 [70, 28] runs IA-32 binaries for the Windows operating system on an Alpha processor, also running Windows. The O/S remains the same, but the ISA is translated.

Java VMs follow yet another model. Java VMs typically run on top of a native O/S. The VM translates Java bytecodes into the underlying ISA, but Java bytecodes do not provide access to system services, like I/O. Instead an *application binary interface* (ABI) provides these services through a set of standardized function calls. The VM translates ABI calls to the equivalent system calls on the underlying O/S. This review focuses on co-designed VMs, such as FX!32 and Java VMs, that perform some form of ISA translation.

2.1.1. VMs for High-Level V-ISAs

To enhance portability new languages have often been compiled into an intermediate representation. Applications are distributed in this form and are translated into the native code of the local machine either when the application is installed, when it first executes, or each time the application executes. The intermediate representation forms a *high-level V-ISA* for an abstract machine that can only be executed by a VM. The Java bytecode architecture, the vehicle for this research, falls into this category.

P-code for Pascal originated this approach [96]. P-code was developed from the internal representation of an early Pascal compiler for the express purpose of porting Pascal to a wide variety of systems. P-code was commonly executed using an interpreter-based VM. Wirth, the creator of Pascal, credits the wide dissemination of Pascal to this innovation [137].

The success of P-code made this a standard technique in language design research. Sometimes, a new language was accompanied by a new high-level V-ISA and a new computer design. This development methodology approaches the co-designed VM concept. However, since the dominant mode of execution was direct execution using micro-code, the I-ISA and V-ISA were still the same or nearly so. The Western Digital Microengine took this approach for Pascal P-code. The Lilith computer directly executed M-code for Modula-2 [136, 97].

The Alto computer from Xerox PARC supported micro-code execution of several bytecode based virtual ISAs [82, 121] for Mesa, Smalltalk, and LISP, as well as a separate instruction set for BCPL. The micro-code on the Alto could be reprogrammed on the fly, depending on the language in which the application is written. The micro-code forms an I-ISA that supports multiple V-ISAs. However, the translation is done in hardware.

The Symbolics machines [93, 94] are early examples of true co-designed VMs. Designed primarily for LISP, the Symbolics line of computers used "a combination of hardware and firmware." [94] to execute a stack-based V-ISA based on 17-bit instructions. As a result, Symbolics was able to change the underlying architecture across machine generations. Hardware support for LISP included tagged memory (two bits per word), automatic type checking and conversion, acceleration for function calls, direct hardware implementation of basic common LISP functions, and a variety of support for garbage collection, which is discussed in Subsection 5.1.3.

As RISC architectures became prevalent, researchers began to study how virtual machines could capitalize on this style of architecture. This led to projects such as SOAR [130] and Mushroom [139, 138], both targeting Smalltalk. These systems provide dynamic compilation from Smalltalk bytecodes to the native co-designed instruction set. Special hardware supports type checking with tagged memory, garbage collection with software-managed caches, object-oriented heap management with object-oriented memory hierarchies, and function calls with register windows. Tuning the virtual machine compiler and the RISC I-ISA together enhances performance in ways that would not be possible without co-design.

The Self-93 system [69], although not co-designed, significantly advanced the state of the art of virtual machines by combining a number of previously studied techniques into an efficient system. Self is a very pure, simple, object-oriented language similar to Smalltalk. The Self-93 system executes Self bytecodes on a SPARC processor.

The Self-93 VM contains no interpreter. All code is compiled with a simple compiler when first invoked. Frequently executed methods are then recompiled at a higher optimization level. This profile-driven reoptimization of program hot spots is a major advancement for virtual machine software. Optimizations generally focus on eliminating type checks and method invocations, which are very frequent in the Self language. Self-93 can also uncompile methods to support debugging features and other rare events.

More recently, Sun Microsystems developed the MAJC processor for Java [91]. MAJC is a co-designed virtual machine that runs Java bytecodes on a VLIW processor using VM software based on Sun Microsystems' HotSpot VM [92, 10]. A MAJC chip contains two cores that directly share a dual-ported L1 D-cache. Each core supports coarse-grain multithreading as well.

MAJC exploits CMP and the high-level semantics of Java to implement a form of speculative multithreading called *space-time computing* [92]. When MAJC encounters a function call, it can execute the code following the call in parallel with the function itself. The second processor executes the code following the call in a speculative thread using techniques similar to Multiscalar [115] and dynamic multithreading [4]. The semantics of Java greatly simplify the implementation. Java allows the call stack frames to be handled specially, makes return values explicit and identifies stores to the heap so that write-after-write hazards can be detected and corrected.

The AS/400 line of computers takes a static approach to binary translation [116]. Application programs are distributed as a *program template*. Before a program is run, an AS/400 platform compiles the program template into an executable binary suitable for that platform. The success of this approach has been demonstrated by IBM's migration of the AS/400 to the PowerPC architecture [147, 35].

2.1.2. VMs for Low-Level V-ISAs

The previously discussed high-level V-ISAs resemble high-level languages in bytecode form. In contrast, low-level V-ISAs resemble instruction sets designed for direct execution. For instance, IA-32 is a common choice for virtualization because it is popular, and because virtualization may greatly simplify the implementation of the CISC architecture.

The System/38 represents a transition from high-level to low-level V-ISAs [18]. System/38 dynamically converted program code into internal micro-code before it was executed. Hiding the details of the micro-code from the virtual architecture provided flexibility for future innovations. The V-ISA for the System/38 was designed to be virtualized and contained both high-level and low-level features. Most application computation took place in *spaces*, dynamically allocated arrays of bytes, using instructions similar to other ISAs of the time. However, many system level objects were manipulated with high-level instructions.

The DAISY (Dynamically Architected Instruction Set from Yorktown) project at IBM's T.J. Watson Research Center articulates the microarchitect's desire to improve performance using innovations incompatible with a standard ISA [55]. The goal of DAISY is to enhance instruction-level parallelism using VLIW architectures that are incompatible with the architecture being addressed.

DAISY contains a kernel—unseen by the virtual architecture—which translates instructions from the virtual ISA to the VLIW architecture. V-ISA code is translated when it is first reached and cached in a *code cache*, a separate region of memory not visible to the V-ISA. Aggressive VLIW scheduling provides improved performance. All aspects of the emulated architecture are preserved, including precise exceptions and self-modifying code semantics; even operating system kernel code is emulated. Hardware support is provided for unusual aspects of the V-ISA, such as different floating-point formats, status and comparison flags, and the ability to access subsections of a register. Target architectures include IA-32, S/390, PowerPC, and Java [54].

The Crusoe processor from Transmeta [62, 78] is similar in concept to the DAISY project. Crusoe transparently executes the complete IA-32 instruction set on a co-designed VLIW processor. Crusoe incorporates a number of architectural features to enable aggressive scheduling in support of the VLIW architecture. For example the *gated-store buffer* [135] holds stores until it is assured that no exceptions have been thrown. Instructions can be scheduled aggressively because stores in the buffer can be squashed to roll back architected state to a known-good point.

FX!32 also treats IA-32 as a V-ISA. FX!32 transparently executes IA-32 binaries on Compaq Alpha processors [28, 70]. Given an Intel IA-32 binary, FX!32 interprets the IA-32 instructions during the first execution to provide profile information. An optimizing compiler then translates the IA-32 code into high-performance Alpha code, and transparently caches the compiled code on disk. Later invocations of the application use the optimized code. As profile information accumulates from successive runs, FX!32 re-optimizes the application. FX!32 works with well-behaved binaries on the Windows operating system. An Alpha version of Windows runs natively on the Alpha processor, so calls to the standard ABI can be executed directly without translation.

2.2. Mechanisms for Thread-Level Parallelism

Figure 2.1 provides an overview of methods used to support multiple threads. Computers have executed multiple threads using software (e.g. multiprogramming), using multiple processor chips or modules (e.g. symmetric multiprocessing), and by supporting multiple threads within one chip or module. Within one chip, multithreading has been supported through chip multiprocessors (CMP) and multithreading.

CMP integrates multiple processor cores on a single die to take advantage of increasing

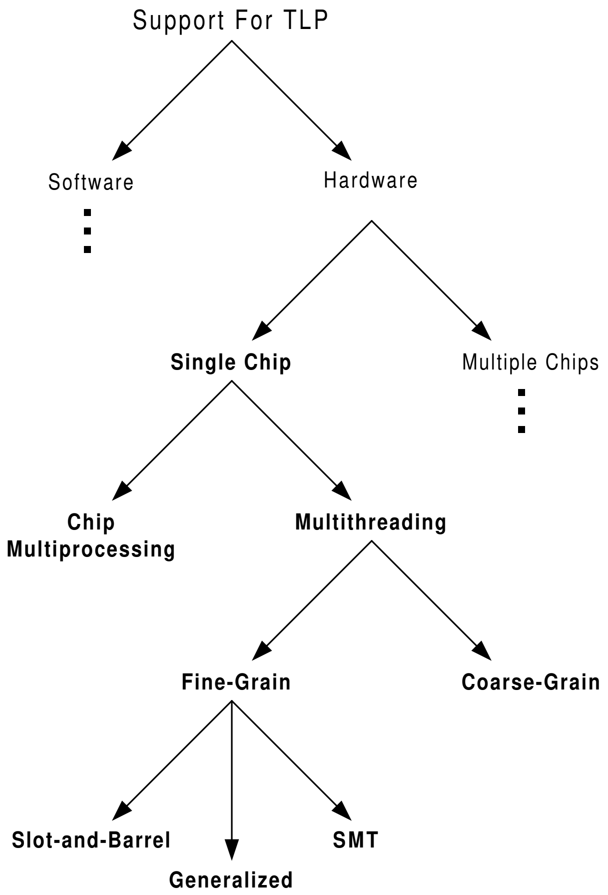


Figure 2.1. Support for thread-level parallelism.

transistor budgets [16, 46, 91, 99]. Studies by Barroso et al. [16] and Olukoton et al. [99] both show that greater instruction throughput can be obtained with several smaller, simpler pipelines then with a single ILP-oriented pipeline, given approximately the same area. Several recent CMP designs have been announced. The Power4 [46] integrates two PowerPC processors on one chip and four chips on one multi-chip module. The MAJC processor [91], discussed above, integrates two multithreaded VLIW cores on one die.

Multithreading has been studied since at least 1958 [53]. Multithreading interleaves instructions from multiple threads through a single processor core without software intervention. Coarse-grain multithreading (CGMT) switches between threads to circumvent long-latency stalls, such as cache misses. Fine-grain multithreading switches between threads on a cycle-by-cycle basis, possibly even fetching instructions from multiple threads in a single cycle. Slot-and-barrel designs cycle through threads in a round-robin fashion, and typically only allow one instruction from each thread to be in the pipeline. More general FGMT techniques use various priority heuristics to select the next thread, and usually allow multiple instructions from the same thread to be in the pipeline. Simultaneous multithreading (SMT) allows instructions from multiple threads to execute simultaneously, particularly in super-scalar out-of-order designs. Multithreading has been shown to cover memory latency [122], cover branch penalties [60] and increase ILP or functional unit utilization [128].

Several early designs were based on the slot-and-barrel FGMT technique [113, 57, 125]. Threads are conceptually arranged on a rotating barrel. Each thread issues one instruction once each rotation. The instruction is guaranteed to complete before the next revolution, so at most one instruction from each thread is inflight at any one time. I used this simple technique in the query engine of the RPA discussed in Subsection 4.1.2.

More general FGMT methods choose among active threads on a cycle-by-cycle basis [124, 67, 6, 104, 124, 75]. Thread scheduling is typically based on a round-robin policy, but modified to select only from threads with an instruction that is ready to issue. Unlike slot-and-barrel designs, multiple instructions from the same thread may be in the pipeline simultaneously. With a more general approach, fewer threads are required to fully utilize a pipeline, and demands on execution latency are reduced, particularly in regards to memory accesses. I use this approach in the service processors used to execute service threads.

SMT is an FGMT technique that allows instructions from multiple threads to issue each cycle [128, 127, 140, 141, 60, 103]. In a typical superscalar out-of-order processor, register renaming can be used to keep the threads separate within a unified instruction window. SMT has the advantage of increasing ILP within the instruction window, as well as tolerating I-cache misses, D-cache misses and branch mispredictions [128]. SMT has also been a popular addition to recent CPU designs. The Pentium 4 incorporates 2-way SMT [89]. The Alpha 21464 was also to support 4-way SMT [45].

Coarse-grain multithreading (CGMT) simplifies the pipeline design by switching between threads less frequently [122, 3, 40, 133]. CGMT keeps multiple contexts on-chip with switching overheads of a few cycles to a few dozen cycles. This technique has been popular in message-passing-based multiprocessors. A context switch is usually caused by a message arrival, a cache miss or failure to obtain a lock. The PowerPC AS processors use 2-way CGMT to improve commercial workload performance [23, 147]. CGMT is preferred over FGMT because it has less impact on core design and more effectively supports thread priorities.

Other hybrid mechanisms for exploiting ILP and TLP exist, such as Hydra [98], a Multiscalar-like [115] paradigm oriented toward multiprocessing. A processor is formed from a series of distributed cores. These cores may execute independent threads or be combined to execute speculative tasks from a single thread. Spawning of tasks is software controlled, and the hardware provides a mechanism to allow speculative memory accesses to be rolled-back.

2.3. Work Related to Service Threads

Service threads are used by VMs to improve application performance and support language features. At least two Java VMs have put service threads into practice. The Jikes research VM [12] creates multiple threads for summarizing profiles and for compilation. The HotSpot VM from Sun Microsystems also supports parallel dynamic compilation [92]. In addition, service threads are often used in the form of concurrent garbage collection. This is discussed in Subsection 5.1.1.

The Instruction Path Coprocessor (I-COP) [34] replaces hardwired optimizing trace-cache fill units [72, 59] with a specialized coprocessor. The 4-way VLIW processor receives complete execution traces containing instructions and values from the processor. This is used to generate traces for the trace cache, performing optimizations such as linked-list pointer prefetching and stride prefetching. The I-COP essentially performs profile analysis and dynamic optimization using a service thread on specialized hardware.

Simultaneous Subordinate Microthreading (SSMT) [27] uses multiple *micro-threads* to improve single-threaded application performance. SSMT executes micro-threads in parallel with the application using SMT. Subordinate threads are written in microcode, and have direct access to the underlying microarchitecture. To show potential, branch prediction is performed by executing custom prediction routines in parallel with micro-threads. Off-line profiling identifies branches for which the hardware gshare predictor is markedly inferior to the PAg [142] predictor managed by the subordinate threads. The compiler explicitly spawns subordinate threads for those branches that will be aided by the PAg predictor. Predictions are computed by the subordinate threads, and communicated back to the hardware through a prediction cache.

Assisted Execution [117] uses *nanothreads* in conjunction with SMT in much the same way. Unlike SSMT and this work, nanothreads share register state with the application thread. Nanothreads are either invoked directly by the application, or triggered by hardware events using *nanotraps*. Special-purpose profiling mechanisms are used to provide the nanothreads with information needed to perform a given optimization. Both sequential and stride-based prefetching are implemented using assisted execution.

The previous examples use service threads for computation not directly derived from the application instruction stream. Many works have used service threads to execute small portions of the application early and in parallel. These techniques often target pre-execution of problematic instructions, such as loads with high cache miss rates, or branches with high misprediction rates.

Farcy et al. [56] use this technique to anticipate the outcomes of branches that are difficult to predict. Following an analysis of the data-flow equations leading up to these branches, they focus on branches that are based on linear arithmetic operations (called *predictable arithmetic expressions*), and loads of such values (called *table traversals*). Such branches are often based on loop index variables. These branches and the instruction slice leading up to them are selected by hand using detailed profile information.

The selected branches are duplicated and executed speculatively using a second thread in an SMT-like manner. Stride-based data-value prediction allows the second thread to jump several iterations ahead of the main thread and resolve branches early. Branch outcomes are communicated to the main thread through a table of prediction queues.

Zilles and Sohi [144] use a similar approach to tackle problematic branches and load instructions. *Speculative slices* are constructed that execute the problematic instruction ahead of the main thread. SMT executes program slices in parallel with the main thread. Branches forward their prediction to the main thread through a table of FIFO queues, indexed by PC. Load instructions automatically prefetch data for the main thread. Slices are constructed by hand, though a technique proposed by Roth and Sohi [106] is suggested for automation.

To be effective, the schemes proposed by Farcy et al. [56] and Zilles and Sohi [144] must execute the slice in a timely manner, ahead of the main program. To allow the slice to execute early enough, Farcy et al. use stride-based data-value prediction. Zilles and Sohi use two different techniques. First, whenever possible the thread is launched well before the problem instruction. Second, since the slice is speculative, it is only used to improve performance, it can be heavily optimized with optimizations that are unsafe. A simple example is eliminating instructions for computing loop exit conditions. A key result here is that speculative slices are often small and handle multiple problematic instructions.

Collins, et al. [36] describe a similar technique targeting problematic load instructions. They study the effects of speculative precomputation on multithreaded statically scheduled VLIW processors, in contrast to the other works reviewed in this section. Collins, et al. propose a method of automatic construction of slices based on functional simulation of the program.

Data-driven multithreading (DDMT) [106] is also used to target problematic load and branch instructions. DDMT executes *data-driven threads* (DDTs) in parallel with the main thread using SMT. A DDT is a trace-like [105] slice of instructions leading up to a problematic instruction. Instructions selected for the slice need not be statically or dynamically contiguous. DDTs execute instructions straight through from beginning to end without following any form of control flow. A DDT is forked upon approaching a problematic instruction with the intent that the DDT reaches the problematic instruction ahead of the main thread. The DDTs out-pace the main thread because they fetch only a small subset of the instructions that are fetched by the main thread.

The DDT duplicates instructions from the main thread. However, due to *register in-*

tegration, a key enabling technology, the processor need not execute any instruction more than once. Register integration uses register renaming to map the output of the duplicated instruction to the output of the original instruction.

DDTs can be constructed transparently within the processor microarchitecture, or using profile-driven compiler technology. In this study, DDTs are constructed automatically from detailed traces of the program execution. Heuristics based on the *fetch-constrained-data-flow-height* (FCDH) select DDTs that are likely to be profitable.

Slipstream processors [119] extend speculative slices to the entire program. The *advanced stream* (A-stream) executes a reduced copy of the program, which is possibly incorrect. The A-stream sends control-flow and value predictions to the *redundant stream* (R-stream), which executes the complete program and validates the execution of the A-stream. The program executes both streams faster than either alone. The A-stream runs faster because it executes fewer instructions. The R-stream runs faster because it has the advantage of the highly accurate branch and value predictions from the A-stream.

To reduce the A-stream, hardware-based data-flow analysis and prediction are used to find instructions that likely do not need to be executed. Such instructions include highly predictable branches, silent stores [85] and silent register writes. Instructions that feed eliminated instructions can also be transitively removed in many cases.

CHAPTER 3

THE RELATIONAL PROFILING ARCHITECTURE

Hardware profiling mechanisms have traditionally been developed using a bottom-up approach. Hardware designers place counters at various places in the design to count microarchitectural events. The primary use for these mechanisms has been performance verification of the CPU design. While these mechanisms can be used for software development, the interfaces are often complex and cumbersome [17, 1]. To better support profiling for software tuning, interfaces to event counters have been generalized. Profiling support has been added to operating systems, and general profiling tools have been developed [8, 86, 2].

Because profiling is critical for exploiting TLP within a VM, we take a top-down approach to hardware-assisted profiling. The goal is to develop mechanisms flexible enough not only to satisfy known applications for profiling, but also future applications that will likely develop as VMs evolve.

The *relational profiling model* was developed after examining the types of information that needed to be collected. Profiling in the relational model is analogous to performing queries on a table of event and instruction data. Given such a model, the next step was to develop a profiling architecture that permits efficient expression of such queries, does not lower application performance and can be efficiently implemented. These goals have been achieved by the relational profiling architecture (RPA).

The RPA selects individual instructions, and collects specific information for these instructions. Selection criteria include instruction op-codes, per-instruction profile tags, and the contents of the collected information. The RPA hardware collects the requested information, producing a stream of standard-format *profile records*. These records are passed to service threads through a novel shared-memory communication mechanism. Service threads read these messages to perform optimizations and other support tasks in parallel with the

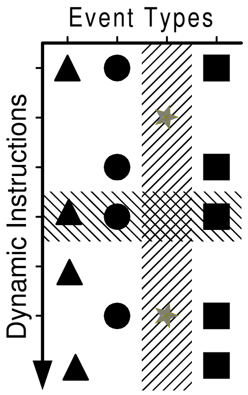


Figure 3.1. The relational profiling model views instructions and events as a table.

main computation thread(s).

3.1. The Relational Profiling Model

The purpose of relational profiling model is to guide the development of the architecture by providing a uniform model for a wide range of information collection. The model is intended to be simple and general, not detailed or specific.

The relational profiling model conceptually organizes profile information in a table like a relational database. See Figure 3.1. Each row represents a dynamic instruction; each column represents an associated piece of information (such as the instruction's PC) or possible event (such as a branch misprediction). As the processor executes instructions, rows are conceptually added to the bottom of the table. As instructions flows through the pipeline the fields in each row are filled in, yielding a complete profile record by the time the instruction commits.

Profiling, then, is analogous to performing a query on this table. For instance, collecting information about branch mispredictions would involve selecting all rows (dynamic instructions) for which a branch misprediction event occurred.

This leads to two basic forms of queries.

1) Instruction queries. "For certain instructions, what events occurred?" These queries select rows from the table. To collect this information, the profile mechanism metaphorically follows an instruction through the pipeline, collecting information regarding its behavior. This is similar to ProfileMe [41].

2) Event queries. "For some type of event, what instructions are involved?" These queries select columns from the table. To collect this information, the profile mechanism observes some point in the pipeline, recording information about instructions that flow past. This is similar to the counter-based profiling mechanisms common in processors today. In contrast to counter-based methods, however, the relational model can provide detailed information about specific dynamic instructions, such as the PCs of all load instructions that miss the L2. Nevertheless, hardware counters may sometimes be useful as an efficient summarizing mechanism.

The development of a profiling task starts with a conceptual query based on the relational model (Figure 3.2a). This is translated into an assembly language query by the developer (Figure 3.2b). This assembly language form is compiled into a profiler binary, which is loaded into the RPA hardware (Figure 3.2c), for instance during VM startup. Once enabled, the RPA hardware profiles the thread executing on the processor, efficiently collecting the records specified by the conceptual query (Figure 3.2d). These records are passed to service threads that process the records, producing the desired profile information (Figure 3.2e). Dynamic construction and modification of profile queries is possible, but this has not been studied.

While the relational model provided guidance for the RPA, actually building a relational table containing all data for every instruction is impractical. Therefore the RPA is designed to reduce buffering and bandwidth by filtering information not of interest.

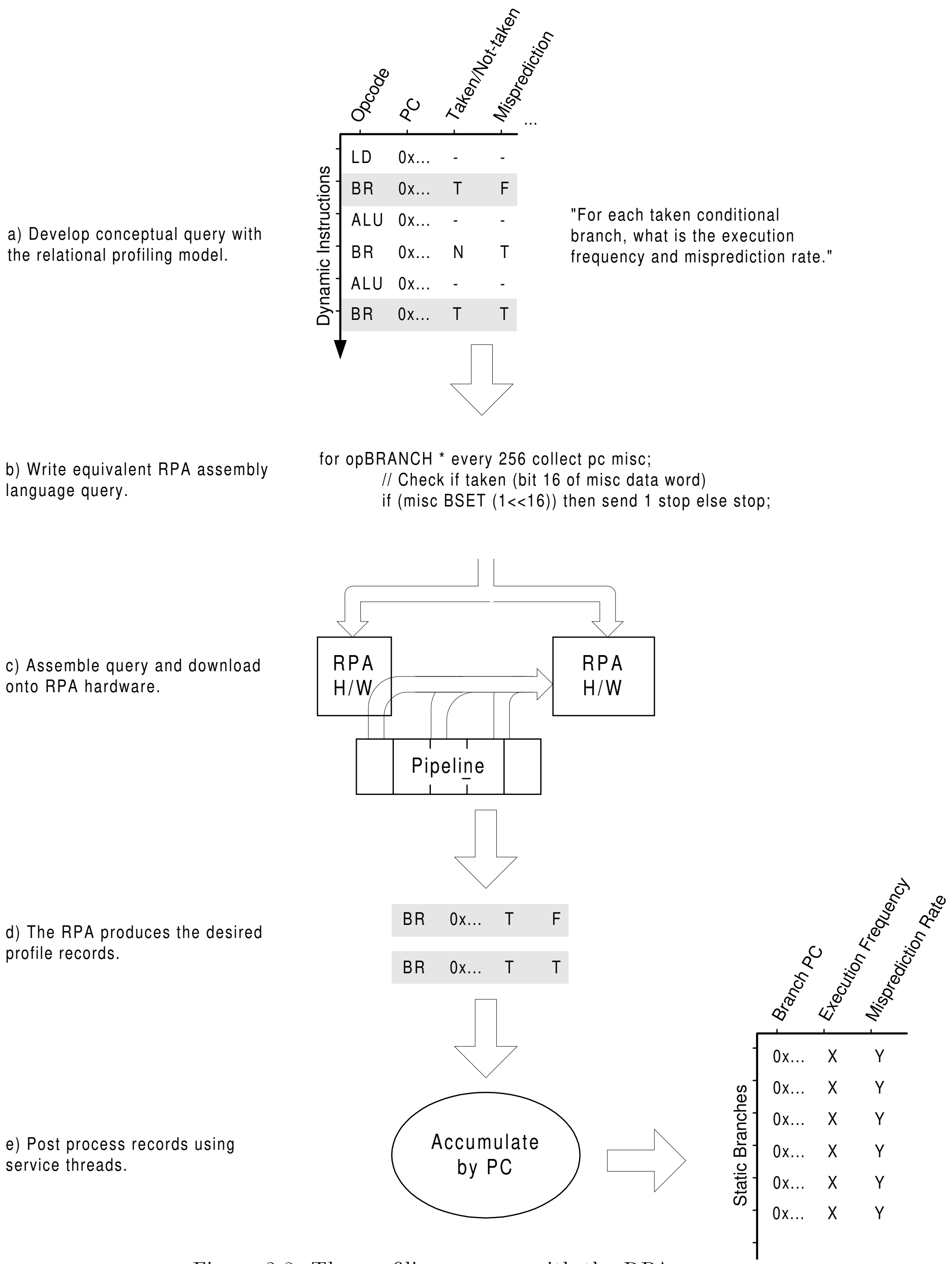


Figure 3.2. The profiling process with the RPA.

```
a)  for opBRANCH * every 256 collect pc misc ;
         send 1 stop ;

b)  for evL2MISS * always collect pc res2 ;
         send 2 stop ;

c)  for opSTORE 1 always collect op1 op2 op3 ;
         if op1 <> 0 then send 3 stop else stop ;

d)  for opBRANCH * op LOAD * opSTORE * opALU *
             opMULT * opFLOAT * opSYS *
             every 1024 collect pc rrate ;
         send 4 stop ;
```

Figure 3.3. Example RPA assembly language queries.

3.2. The RPA Assembly Language

The RPA collects information. The RPA assembly language breaks this task into one or more queries, like those shown in Figure 3.3. An RPA query 1) lists the information to be collected, 2) specifies a rate at which the information should be collected, 3) describes selection criteria against which a record should be checked, and 4) indicates how the selected information should be communicated back to software.

Each query begins with a *query header*, and is followed by one or more *query clauses*. Query headers are identified by the *for* keyword; all other statements are clauses. Each query in Figure 3.3 contains a single clause, the common case.

The query header indicates which instructions to examine, what information to collect and how often to collect it. The types of instructions to profile are listed following the *for* keyword. Instructions are divided by op-code into the eight classes shown in Table 3.1. For example, the query in Figure 3.3a specifies that conditional branches should be profiled.

The RPA further classifies instructions using a two-bit profile tag per instruction in the

Table 3.1. Instruction profiling classes.

Mnemonic	*Instructions*
opJMP	Unconditional jumps
opBRANCH	Conditional branches
opLOAD	Load instructions
opSTORE	Store instructions
opALU	Simple arithmetic and logical instructions
opMULT	Multiply/divide instructions
opFLOAT	Floating point operations
opSYS	SYSCALL, BREAK, etc.
evL2MISS	L2 misses
evITLBMISS	Instruction TLB misses
evDTLBMISS	Data TLB misses

program binary being profiled. This yields a total of 32 instruction classes. Each query may profile any or all classes. The VM paradigm allows this extra two-bit field to be added to the implementation ISA. An alternative is to add additional hardware tables to hold software-controlled classification information.

The query header can also select certain processor events to profile. Event profiling selects instructions based on the events that instructions cause, rather than by op-code. Theoretically, event profiling could be dispensed with entirely by collecting instruction-based information and then selecting only those instructions with the desired event. For rare events this leads to very long profile times, however. Hence the proposed RPA specifies event profiling for rare events like I-TLB, D-TLB and L2 cache misses. Event queries cannot determine if an instruction needs to be profiled until the event occurs (e.g. until the load misses the L2 cache). This is likely to be too late to collect information from earlier pipeline stages, so some information may not be possible to collect with event profiling. However, at least the instruction PCs, cycle time stamps and effective addresses are expected to be available.

For example, the query of Figure 3.3c only profiles stores with a profile tag of 1. This query is used by the concurrent GC algorithm (Chapter 5), which only needs to profile those particular store instructions that write references to the heap. The VM tells the RPA which stores to profile by setting the profile tag of such stores to 1.

For the specified instruction class(es), the query header indicates the information to be collected and a sampling rate. For traditional profiling tasks, random sampling can reduce the rate at which profile information is produced, as in Figure 3.3a and d. If the profiled event is rare, as in Figure 3.3b, this may not be necessary. The *always* keyword ensures that every event is profiled. This will effectively reduce the profiling latency, allowing the VM to react faster to problematic conditions.

In other cases, random sampling is completely inappropriate. The *always* keyword enables the RPA to be used for instrumentation, where the information is required for correctness. The GC algorithm described in Chapter 5 must observe every reference store. Hence, the GC query of Figure 3.3d does not use random sampling.

The information to collect is listed at the end of the query header using mnemonics from Table 3.2. The example in Figure 3.3b collects the PC and Result 2 (the effective address) of L2 data misses. Collectible data includes both architected and implementation information. Each mnemonic represents one 32-bit word of information that is collected and packed into a record by the RPA. The proposed RPA limits the information collected to seven words per record. This limit is somewhat arbitrary, but this appears to be large enough to handle all profiling tasks without producing overwhelmingly large records.

Following the query header, query clauses compare records to selection criteria and communicate selected information back to service threads. Query clauses execute sequentially in a manner similar to branch instructions. Like branch instructions, query clauses can be arranged into decision trees to compute arbitrary Boolean functions. Processing of a record terminates when an explicit *stop* keyword is reached.

Table 3.2. Information the RPA can collect.

Mnemonic	*Information*
pc	Instruct PC
thread	Thread ID
op1, op1up, op2, op2up, op3	Input operand values
res1, res1u res2	Output operand values
misc	Exceptions, branch outcomes, branch mispredictions
ftime	Cycle when instruction fetched
rtime	Cycle when instruction retired
frate, drate, irate	Fetch, dispatch and issue rates
rrate	Retire rate
wlat	Execution window latency
elat	Execution latency

Table 3.3. Query clause comparison types.

Mnemonic	*Comparison*
<=>	All unsigned integer magnitude comparisons
BCLR	True if bits under mask are zero
BSET	True if bits under mask are one
FILTER 2/4/16/256	Random filter. True with probability 1 in 2/4/16/256.
TRUE/FALSE	Always true/Always false

Each query clause can perform one or two comparisons to check for desired properties within the record. If the comparison(s) match, the clause branches to another query clause. Otherwise, execution falls through to the next sequential query clause. The available comparisons, shown in Table 3.3, are the standard relational operations, tests for set or cleared bits and further random sampling. Comparisons operate on 8, 16 or 32 bit unsigned integer values so both the size and location of the data within a record are encoded in the query.

Figure 3.3c provides a simple example. Because stores of null references do not need to be examined by the GC algorithm, the single query clause of Figure 3.3c only sends records if operand 1, the stored reference, is not null (zero).

Query clauses perform *profile actions* to communicate collected information back to VM software. The query clause executes an action if the comparison(s) are true.

A *counter* action increments a counter embedded in the query clause itself. This allows software to construct custom counters for arbitrary events using the query engine. When the counter is close to overflowing, the thread being profiled is interrupted.

An *interrupt* action interrupts the thread being profiled. The interrupt may be synchronous or asynchronous. For synchronous interrupts, the profiled instruction retires, but subsequent retirement stalls until the query is completed.

This research focuses on the *send* action, indicated by the *send* keyword in Figure 3.3. Send actions copy selected records to service threads for further processing. The communication occurs via FIFO message queues held in shared memory. Queues contain 8-word (32-byte) records, which are 32-byte aligned. The address and size of the queues are both configurable.

To send a record, the RPA writes the seven words of the profile record, and then the eighth *ready word*. The ready word indicates that the record is available, and specifies which query clause produced the record. To read a record, service threads poll the ready word, read the record, and clear the ready word using normal loads and stores. When reading records from multiple queries, service threads use the ready word to determine the type of each record, as demonstrated in Subsection 6.8.1.

The RPA must not reuse a slot in the message queue until the service thread has read the record. Although the ready words could be used for this purpose, the *buffer read status word* provides a more efficient mechanism. Service threads periodically store the number of messages read into the buffer read status word. This word is examined periodically by the RPA. The RPA uses the number of messages read, along with the number written (which it knows), to determine if space is available for another message. This system reduces the polling required by the RPA, over using ready words alone.

The RPA can manage multiple message queues, sending records from different queries to different queues. This way service threads only receive information related to their task, often eliminating the need to determine what type of message has been received. In Figure 3.3, each query sends different records to different service threads, as indicated by the integer identifier following the *send* keywords.

The RPA can also use multiple queues to increase the processing bandwidth for records from a single query. Collections of queues used in this manner are called a *pool*. The RPA disperses records to all the queues in a pool so that multiple service threads can efficiently consume the records without the synchronization overhead that would be required by a single common queue. The RPA also performs load balancing among the queues in a pool, sending new messages to the queue that is most nearly empty. The integer identifier following the *send* keyword specifies the destination pool for each record.

Figure 3.4 shows a simple example. The query of Figure 3.3a randomly samples one branch out of 256, producing a low-bandwidth record stream. These records are sent to Pool 1, consisting of a single message queue accessed by a single service thread. The store profiling in Figure 3.3c cannot use random sampling, and therefore produces a higher bandwidth stream. For this reason, the profile records are sent to Pool 3 containing two message queues read by two service threads.

3.2.1. Consuming Messages Efficiently

Service threads make use of two principles to consume messages efficiently. First, if the queue is nearly empty, efficiency is not important. The consuming service thread is obviously keeping pace with record production. On the other hand, when there are many records in the queue, performance can be improved by consuming multiple records at once.

Figure 3.5 shows simplified code illustrating the technique generally used by service threads to read messages. The code assumes the queue contains an even multiple of four

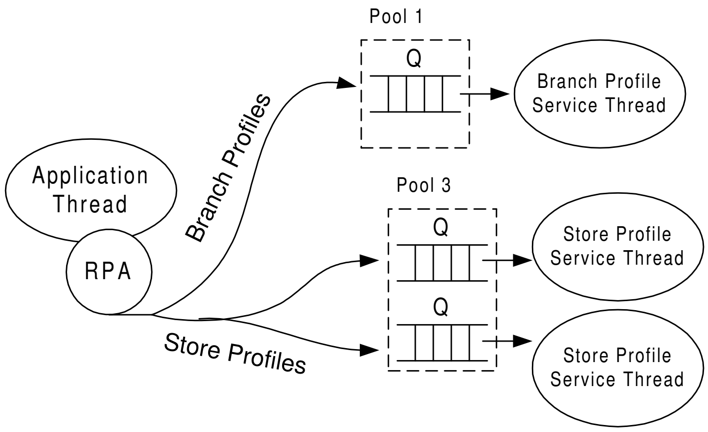


Figure 3.4. The RPA sends profile records to message queues in pools.

records; I generally use queues with 32 records.

The code contains three distinct sections. When the queue is empty (Section A) the service thread continuously polls the ready word of the next available record. Since the queue is empty, this is a good place to check for mode changes, such as stopping the service thread or changing the service task. The garbage collector uses this feature.

Section C is used when only a few messages are available. Records are consumed one at a time. The ready word is cleared, and the buffer read status word is updated for each record. The overhead in this mode is about six instructions per record, plus one instruction to read each data word of each record.[1]

If the queue starts to fill, the service threads will transition into Section B. Section B efficiently consumes four records at a time, eliminating the intermediate checks and updates. The record alignment check $((nextRec\&3) == 0)$ makes sure that the end of the queue will not occur within the four records; there is no need to check for wrap around. By checking that the fourth record is available, the code assures that all four records are available.[2] The

[1] The Consume(...) function represents inlined code. There typically is no call overhead.

[2] The RPA guarantees that the slots in the queue are filled in order. This algorithm will be problematic on architectures with weak consistency models, such as PowerPC and Alpha.

```
Record *startRec;         // Address of the first record in queue
Record *endRec;           // Address of the end of the queue
Record *nextRec;          // Next record to consume
int localBRS;             // Number of records consumed
int *bufferReadStatus;    // Address of buffer read status word

while(true) { // Main consumer loop

a)      // No records available
        while(!nextRec->ready) {
             // Stop/start/mode change
        }
        // At least one record is available
        if (nextRec & 3)==0) { // Check alignment
             while (nextRec[3].ready)) {

b)                  // Many records available
                    // Consume four records
                    Consume(&nextRec[0]); nextRec[0].ready = 0;
                    Consume(&nextRec[1]); nextRec[1].ready = 0;
                    Consume(&nextRec[2]); nextRec[2].ready = 0;
                    Consume(&nextRec[3]); nextRec[3].ready = 0;

                    localBRS+=4; (*bufferReadStatus) = localBRS;
                    nextRec +=4;
                    if (curBuf == endBuf)
                         curBuf = startBuf;
             }
        }
c)      // Only a few records available
        // Consume one record
        Consume(&nextRec[0]); nextRec[0].ready = 0;
        localBRS+=1; (*bufferReadStatus) = localBRS;
        nextRec +=4;
        if (curBuf == endBuf) {
             curBuf = startBuf;
        }
} // End main consumer loop
```

Figure 3.5. Service threads efficiently consume records four at a time.

buffer read status word and the *nextRec* pointer are updated only once for each group of four records. The total overhead for processing all four records is about ten instructions, or 2.5 instructions per record, plus one instruction per accessed record word.

3.2.2. Examples

The four queries shown in Figure 3.3 illustrate how the RPA can specify different profiling applications.

Example a) Edge Profiling—Edge profiles, counts of how often conditional branches are taken and not-taken, are one of the most useful types of profiles, enabling or improving several important optimizations [20, 26, 143]. These profiles are also relatively easy to collect. Figure 3.3a shows the RPA assembly for edge profiling. The query header indicates the PC and branch outcome (taken/not-taken) result as well as a branch misprediction flag (*misc*) for branch instructions are collected. Random sampling is used to select one out of 256 branches. This query performs no comparisons. It simply sends the message to a service thread that tabulates the information.

Example b) Prefetching—It has been proposed that data prefetching can be done by assistant threads or nanothreads [117] which are essentially types of service threads. An RPA query for performing this profiling is in Figure 3.3b. The RPA is used to monitor L2 cache misses. For each L2 cache miss the instruction PC and effective address are collected. These records are sent to a service thread that executes prefetch instructions on behalf of the application. This is straightforward if the service thread and application thread share the L2 cache. In other situations, a dynamic optimizer running in a service thread could insert prefetch instructions into the binary, or the service thread could configure an existing hardware prefetch mechanism.

Example c) Garbage Collection (GC)—Figure 3.3c shows the RPA query for the low-overhead concurrent GC algorithm of Chapter 5. To perform GC without stopping the application, the GC thread must monitor certain application stores. This is typically done by instrumenting each store with write-barrier code, unfortunately slowing down the application. The RPA instruments all instructions that write references to the heap without adding instructions. In Figure 3.3c, input operand 1, *op1*, contains the value being written. If this value is not null (zero), then the record is passed to GC service threads that execute the write-barrier code.

Example d) Concurrency Metrics—ProfileMe [41] uses paired sampling to estimate concurrency metrics, such as wasted issue slots in the vicinity of a given instruction. The RPA can compute similar concurrency metrics without using paired sampling. For instance, the query in Figure 3.3 can be used to compute cycles-per- instruction (CPI) for individual instructions and regions, such as loops and functions.

The query collects the PC and retirement rate data for all instruction types. Retirement rate data, *rrate* in Figure 3.3d and Table 3.2, contains two basic pieces of information. The first is the number of instructions, n, that retired during the same cycle as the sampled instruction. The second is the number of preceding cycles in which no instructions retired, c. These cycles are attributed to the sampled instruction only if it is the head instruction during these stalled cycles. Hence, $c = 0$ when the sampled instruction is not the first instruction retired after the stalled cycles.

The execution cycles for one instruction sample is computed as $C = c + 1/n$. The total cycles spent executing the instruction can be computed by summing all the samples, and dividing by the sample rate. The CPI for one instruction can be computed as the arithmetic average over all samples collected for that instruction. The total time spent in a region of code can be computed by summing the total cycles for each instruction in the region. The CPI over a region can be computed using an arithmetic average weighted by relative instruction execution frequency.

To locate bottlenecks, the RPA collects not only the number of retirement stall cycles (c), but categorizes stall cycles by the reason for the stall. Example categories are an empty window, the head instruction not yet issued, or the head instruction not yet completed. Because stalls often have multiple causes, information gained from such categorization is usually approximate. The RPA can collect stall information at other stages in the pipeline as well.

3.3. Summary

The RPA, based on the relational profiling model, is a general mechanism capable of performing a wide range of profiling and instrumentation. The RPA can reduce the bandwidth of the profile stream by carefully selecting which instructions are profiled and what information is collected. The *send* action transfers profile records to service threads without interrupting the application thread being profiled. Records of different types can be sent to different threads, and multiple service threads can be harnessed to increase processing power.

Not only does the RPA support a wide range of profiling tasks, but low-cost implementations are possible. The next section describes one possible implementation, evaluating the hardware resources used by the RPA and the performance overhead associated with various profiling tasks.

CHAPTER 4

RPA IMPLEMENTATION AND PERFORMANCE

The previous chapter described the RPA at the assembly language level. This chapter delves deeper. Hardware structures implementing the RPA are described at a machine code and microarchitectural level. This is followed by an investigation of the performance of the RPA.

4.1. Hardware Structures

Figure 4.1 illustrates the implementation of the RPA modeled for this research. The two main structures are the *profile control table* (PCT) and the *query engine*. An RPA binary contains two sections that manage these two hardware structures. The RPA binary is down-loaded into RPA hardware registers using a special *SET_PCR* (profile control register) instruction added to the I-ISA.

4.1.1. The Profile Control Table (PCT)

The configuration for the PCT is derived from the query headers. The PCT selects which instructions are profiled and determines what information is collected. During dispatch it is accessed by instruction op-code class and profile tag for each instruction . Random sampling

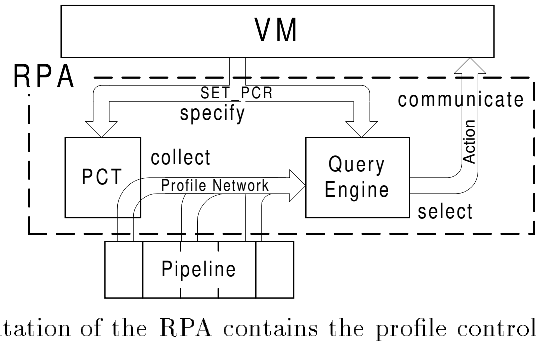


Figure 4.1. An implementation of the RPA contains the profile control table (PCT) and the query engine.

is also performed by the PCT, for instance using linear-feedback shift registers.

The PCT uses two PCRs for each of the 32 instruction classes and the three event types. The first PCR selects the sampling rate and the information to be collected. The second PCR contains the starting *query PC* (QPC), the address of the initial query instruction to be executed by the query engine described in the next section.

As instructions selected by the PCT continue through the pipeline, information is collected by the profile network and buffered. The buffered profile records are then examined by the query engine.

4.1.2. The Query Engine

The query engine is a simple processor capable of performing the comparisons and actions dictated by the query. Each query clause assembles into a single query instruction executed by the query engine. Like query clauses, each query instruction can specify up to two comparisons, a branch and a profile action.

The query engine begins executing the query instructions at the initial QPC location provided by the PCT. Query instructions are executed until terminated by an explicit *stop* annotation within an instruction. The application program continues to execute in parallel with the query engine, and multiple queries may be executed in parallel as well. To simplify implementation there is no guarantee of the order in which separate queries are executed or completed.

The query instructions are encoded in a 64-bit format. The basic two-comparison format is shown in Figure 4.2. The query instruction reads two values from the record. To reduce implementation costs, one even numbered word and one odd numbered word from the profile record are selected. The instruction also includes a 16 bit immediate value. If both comparisons in the instruction match, the specified action is performed and the branch

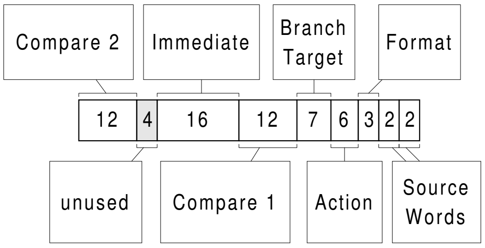


Figure 4.2. Query instructions contain up to two comparisons, a branch and an action.

Table 4.1. Message engine PCRs.

PCR	*Contents*
ME_POOL	Pool that this queue is apart of
ME_ADDR	Base address of queue
ME_SIZE	Size of queue in records
ME_BRS	Address of buffer read status word
ME_BRSVAL	Most recently polled value of the buffer read status word
ME_WRITTEN	Total number of records written

is taken. Otherwise execution falls through to the next sequential query instruction.

The *message engine* handles the *send* actions on behalf of the query engine. The state of each message queue is defined by six PCRs shown in Table 4.1. These PCRs control the size and address of each queue, and the address of the buffer read status word. The current architecture rather arbitrarily sets the maximum queue size at 128 messages. The message engine also polls the buffer read status word. The buffer read status word is polled independently of writing messages using an adaptive algorithm. The message engine polls the ready word more frequently as the message queue fills.

4.2. RPA Implementation and Cost

Two concerns for any profiling mechanism are the hardware cost of implementing it, discussed here, and the performance penalty for using it, discussed in Section 4.5. Although the RPA is a rich and flexible mechanism, implementation costs are low. Table 4.2 lists the

Table 4.2. RPA component costs.

Component	*Size*
Profile Control Table	175 bytes of SRAM, LSFRs
Profile Networks	4 networks
Profile Buffers	8 buffers, 256 bytes of SRAM
Query Engine	Four stage simple pipeline
Message Engine	14 bytes of SRAM per queue

hardware components to the RPA with an estimated size.

4.2.1. The Profile Control Table

The PCT contains about 175 bytes of SRAM (five bytes[1] for each of 32 instruction classes, and 3 events). Records are stored in profile buffers while being examined by the query engine. Section 4.5 shows that eight buffers, 256 bytes total, is enough to minimize the performance overhead.

4.2.2. The Query Engine

The query engine pipeline modeled for this research is simplified by its multithreaded design and its minimal architected state per active query—only the QPC. The design, shown in Figure 4.3, is a very simple four stage pipeline able to execute four queries simultaneously using a barrel-and-slot design [125]. One query instruction is executed for each active query once every four cycles in a round-robin fashion. As the profile buffers fill, the processing power of the query engine naturally increases to one query instruction per cycle. The barrel-and-slot design eliminates all interlocks and dependences in the pipeline as well as branch misprediction penalties.

The first pipeline stage fetches the query instruction from the query instruction array.

[1]This assumes the initial QPC can be stored in a single byte.

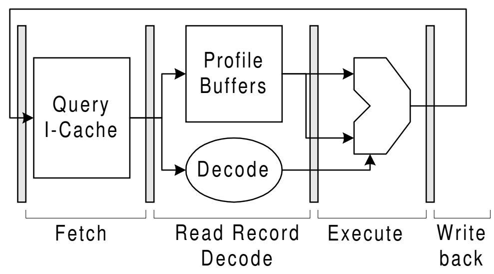


Figure 4.3. The query engine is a simple four stage pipeline.

Query instructions may be stored in memory, or in a special-purpose table constructed out of PCRs to reduce implementation costs. A special-purpose table is modeled in this dissertation. At 512 bytes, this array holds 64 instructions, which appears to be plenty. Decode is performed in stage 2. The record buffers are also accessed in this cycle. Profile buffers are word-interleaved; the query engine accesses one word from the even bank, and one word from the odd bank each cycle. The query ALU performs the comparison(s) in stage 3 using a masked magnitude comparator. The fourth stage selects and drives the next query PC back to the fetch stage. Query actions are also initiated in the write-back stage.

The message engine contains about 14 bytes of state per message queue. The number of queues supported is left open, though this research has never used more than four. The message engine reads records from the record buffers two words at a time using a second dedicated port, and writes them to the in-memory queues. Ideally, the message engine would bypass the processor's L1 caches. However, the simulator used for these studies stores messages into the L1 data cache.

4.2.3. The Profile Networks

Perhaps the greatest implementation concern is the profile network. Fortunately this network is latency tolerant (see Subsection 4.5.5); the profile mechanism can be relatively distant from the core pipeline. However the size and layout impact of this network on the core must be minimized. Though a complete design of this network is beyond the scope of this work, the size of the network will scale at worst linearly with the number of simultaneously profiled instructions. Like the ProfileMe mechanism [42], we model multiple identical profile networks to simultaneously profile multiple instructions. This represents a very conservative design. Several variations and alternatives present themselves.

One alternative is to keep all information with the instruction as it flows through the pipeline, for instance in an extended re-order buffer. When the instruction retires, the needed information is leisurely copied out of the re-order buffer into the profile buffer.

The significant difference between these schemes is *not* where the information is stored (profile buffer or re-order buffer), but how the communication network is allocated. The simple conservative scheme modeled for this research assigns a network to a particular instruction to collect all types of information. The re-order buffer scheme dedicates networks to a specific type or source of data for all instructions. This mechanism can be used without extending the re-order buffer. A network from each data source within the pipeline routes information directly to the profile buffers, bypassing the re-order buffer.

Of course, some information is needed in the re-order buffer anyway. This suggests a hybrid approach. A profile network would not be allocated, if all the collected information is available in the re-order buffer when the instruction commits. If information not readily available during commit is requested, a profile network would be allocated.

Adding instructions to the architecture to explicitly build profile records may also simplify the profile network. Explicit instructions executed by the application would allocate profile records and fill them with data from general purpose registers. Such instructions would issue like other instructions to a dedicated functional unit, which would route data from the input operands to the profile buffers. This scheme, like the re-order buffer scheme, dedicates a network to a specific data source for all instructions. It has the added benefit that issue logic would naturally prevent multiple instructions from trying to use this resource at the same time.

In a particularly simple design, these explicit instructions may be the only way to guarantee information is collected. Automatic instruction selection via the PCT would be used for profiling with random sampling, and explicit instructions would be used as a still low-cost instrumentation mechanism.

4.2.4. Profiling Overhead

Performance penalties arise when the RPA runs out of hardware resources. When the PCT selects an instruction during dispatch, one profile buffer and one profile network must be allocated. Dispatch stalls if either is unavailable. When the instruction retires, the profile network is freed, but the profile buffer remains allocated until the query completes. Section 4.5 shows that four profile networks is enough to minimize stalls due to profiling.

It may be possible to simplify the design or improve performance by dropping random samples when hardware resources are not available. This has been avoided to better evaluate what is required to collect the requested information at the requested rate. One possible concern with this approach is that application behavior would alter the sampling rates of various instructions. For instance, some records from some instruction may always be dropped because resources are never available when it is dispatched.

If the service threads cannot keep up, the in-memory queues will fill. When this occurs, the message engine will stop sending messages, the profile buffers will fill, and dispatch will eventually stall. Hence, it is important that service threads be able to consume messages faster than the RPA produces them.

Poor software design can lead to dead-lock. This and other system software issues are

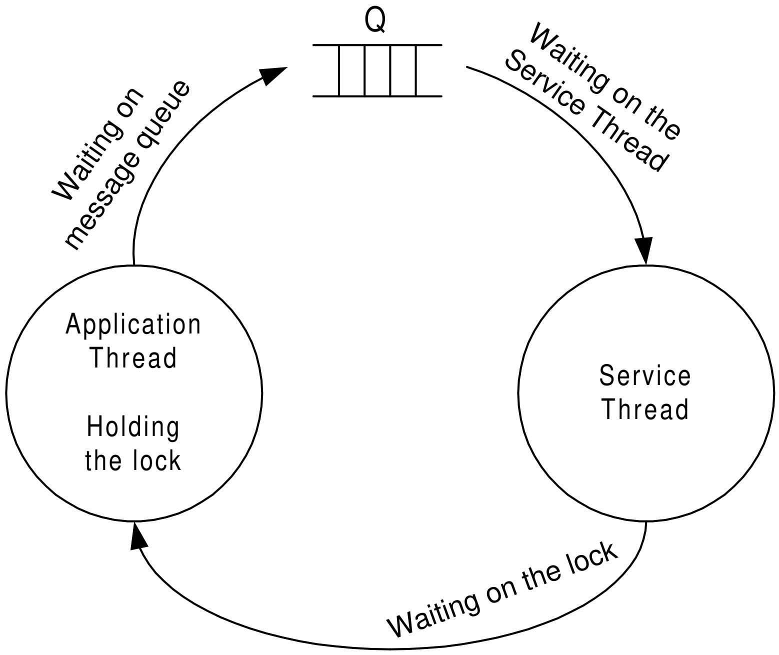


Figure 4.4. A common dead-lock situation with profiling.

discussed in the next section.

4.3. System Software Issues

Development of the RPA revealed a number of interesting interactions between system software, such as O/S code, and the profiling architecture. Although O/S code is not simulated in this dissertation, these issues are addressed briefly here for completeness.

4.3.1. Dead-Locks

Research into the RPA and object co-location (see Chapter 6) uncovered a very general dead-lock scenario. This scenario is thought to apply to a wide variety of hardware assisted profiling schemes, not just the RPA. The dead-lock situation is illustrated in Figure 4.4.

In the simulations of Section 6.10, profiles for object co-location and branches are consumed by a single service thread. Object co-location only profiles code from Java applications. Branches, however, are profiled in all portions of the executable— Java code, the Strata run-time system and C library code.

The scenario begins when the Java application makes a call into the run-time system, which dynamically allocates memory using *malloc*. The *malloc* library is multithreaded, so a lock is acquired for synchronization. Within *malloc*, a branch is selected by the PCT for profiling. Unfortunately, all profile buffers are full, so dispatch stalls.

The service thread consuming the profiles uses a hash table to map instruction PCs to execution counts. While processing a previous profile record, not the record for the branch instruction, the service thread inserts a new PC into the hash table, leading to a call to *malloc*. The service thread attempts to acquire the same lock held by the main application thread, completing the dependency loop resulting in the dead-lock situation of Figure 4.4.

The scenario has a number of interesting facets. First, this scenario could occur with a wide range profilers; the message passing feature is not essential. For instance, the same situation could occur with interrupt driven profiler, such as ProfileMe. Dead-lock would occur if the code handling the profile interrupt makes a call to *malloc*.

Second, the dead-lock can easily occur through locks within standard libraries, not just with locks explicitly managed by the service thread code. In this case, the lock is buried under two layers of library calls.

Third, the profile record that stalls the application thread *does not* require dynamic allocation, and the service thread *does not* lock any locks to process these records. The dead-lock occurs between the main thread and a previous profile record, taken outside of the run-time system while no locks are held. The conditions for the particular dead-lock situation of Figure 4.4 are as follows:

1. A service thread is consuming messages produced by a particular application thread.

2. Both threads can acquire the same lock.

3. While holding the lock, the thread being profiled can produce a profile record consumed by the service thread.

This situation is difficult to avoid in software, without avoiding basic system services like dynamic memory allocation. Hence, we modify the RPA slightly. If an instruction selected for profiling stalls for a long time (300 cycles in our implementation), then the profile can be dropped, provided the instruction is randomly sampled. Generally, profile records are rarely dropped; most of our simulations dropped no samples. Note that this approach is not feasible with interrupt-driven approaches, because the interrupt invokes the profile code before a problem can be detected.

This softens the dead-lock conditions above. Dead-lock can only occur if instrumented instructions, instructions guaranteed to be profiled, are executed while holding the lock. Were this condition to occur, it would likely represent a conceptual flaw in the design of the profile task. One of two general heuristics can help avoid this situation.

1. Service threads consuming guaranteed profile records should not acquire locks *when processing any records*, even records from randomly sampled instructions. This can be done by segregating service threads into those that process guaranteed samples, and those which process random samples.

2. Do collect required samples while holding low-level library locks.

Poor software design can still result in dead-lock. Therefore, as a last resort, an interrupt is needed in case dispatch stalls on a required sample.

4.3.2. Virtual Memory

Addresses of the message queues (ME_ADDR, ME_BRS of Table 4.1) are specified by physical address, not virtual address, so the query engine does not need a TLB to translate addresses, nor be capable of handling protection or page faults[2]. This requires that the VM configure the queues using a system call, and that the O/S pin those pages that contain RPA message queues.

4.3.3. Context Switching

The RPA configuration represents architected state. Therefore, the state of the RPA must be reloaded on context switches. Most of the RPA state does not need to be saved, since it is static. The O/S can make a copy of the static configuration when the RPA is configured for the first time.

The state of the message engine changes with time and will have to be both saved and restored across context switches. For instance, the location of the next free record in the queue changes continuously.

All profile records must be flushed out of the RPA before the queries can be changed. Otherwise, records collected at the behest of the old queries, but still being processed by the query engine, may come under the control of the new queries.

For this purpose the RPA provides a simple mechanism for flushing inflight records. A single PCR that indicates how many records are being held in the profile buffers, the query engine and the message engine. Software polls the PCR until it this goes to zero. The concurrent garbage collector uses this feature to make sure no write-barrier messages are left in the system before collecting objects (see Subsection 5.2.4, Steps 10-17)

In addition, for good performance the thread scheduling algorithm should understand the

[2] Our simulator does not execute O/S code or simulate virtual memory in detail

producer/consumer relationships between threads. Application threads and service threads should be gang scheduled. Running the application with the corresponding service threads switched-out would have disastrous performance consequences. In addition, service threads should be scheduled physically near the thread being profiled, and should be assigned to service processors. Supporting such features within the O/S or the VM is beyond the scope of this research.

4.4. Infrastructure and Methodology

In preparation for presenting simulation results, this section describes the research infrastructure and simulation methodology used. This includes an overview of the features and function of the Strata VM, the SimpleMP simulator, the benchmarks simulated and the sampling methods used during simulation.

Figure 4.5 overviews the infrastructure used for this research. The Strata compiler translates Java bytecodes into PISA assembly code, the instruction set used by the SimpleScalar simulator [24], of which SimpleMP is a derivative. The assembly code is linked together with the Strata run-time system and standard libraries to produce a stand-alone executable. This executable is simulated by the SimpleMP simulator.

RPA queries are assembled into binary form using an assembler developed using the ANTLR tool [101]. This binary loaded into the PCR's by SimpleMP. Even though the VM should perform this initialization step itself, results are not altered because the simulator executes past VM initialization before taking measurements, as described in Subsection 4.4.5.

4.4.1. The Strata VM

The Strata VM consists of two portions. The Strata compiler compiles Java bytecodes to SimpleScalar PISA assembly code, or to aid in development, SPARC assembly code. The

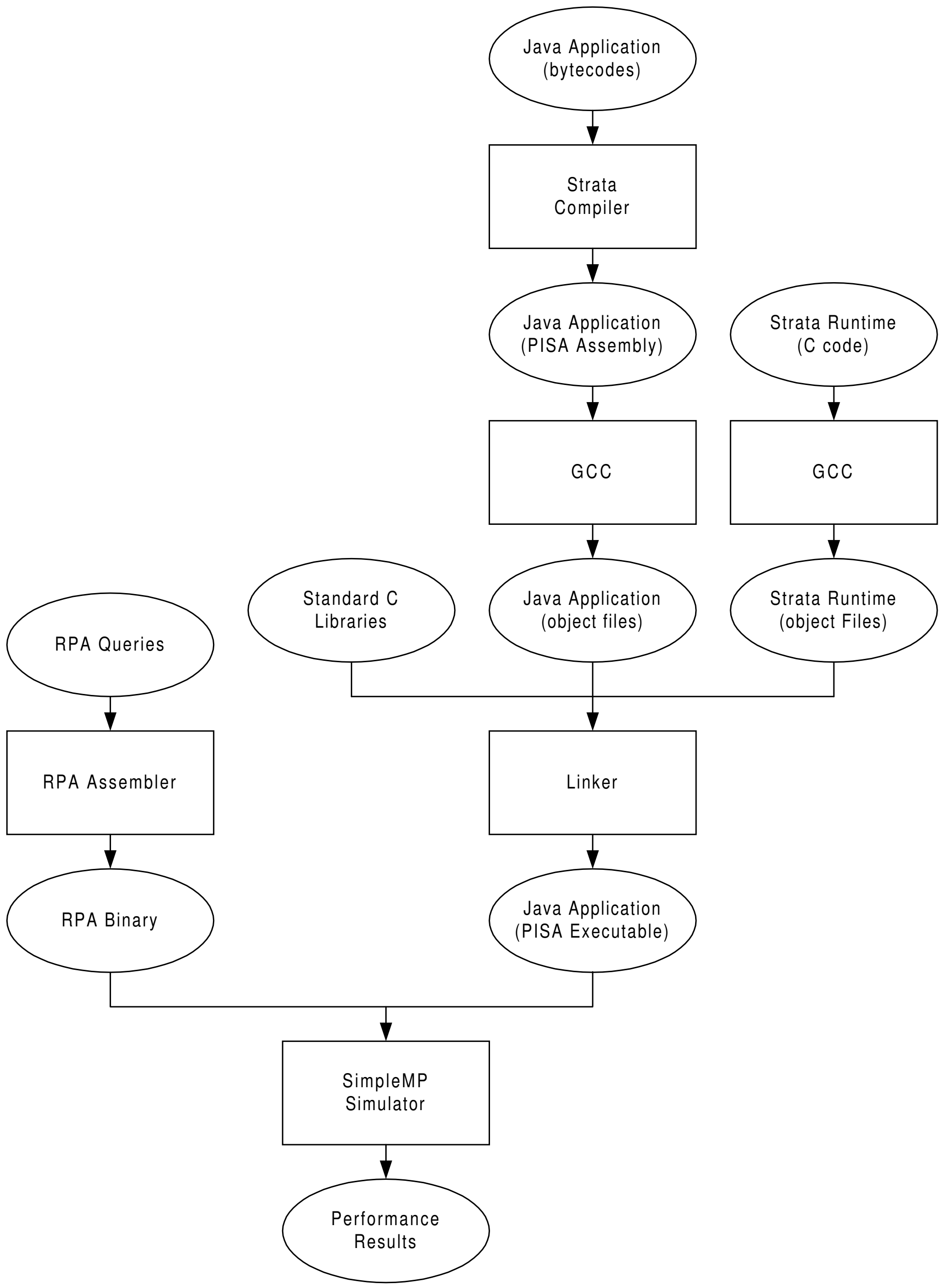


Figure 4.5. Infrastructure used for this research.

Strata compiler is itself written in Java, and forms one of our better benchmarks.

The Strata run-time system contains the garbage collector, I/O primitives, the Java Native Interface (JNI), and Java primitives such as hash codes and cloning. The Java run-time system is written in C.

The Strata Compiler

Rather than develop a complete VM, Strata statically compiles Java into either SPARC assembly code of PISA assembly code. This provides a simple and flexible system for research, while reducing the effort of building and maintaining the system. Essentially, only the execution of the run-time system and dynamically optimized code is simulated. Other phases of execution, such as compilation and interpretation, are not simulated. In a well-tuned VM, these phases of execution should not dominate execution time. We expect future VMs to perform compilation in parallel with application execution (using service threads), and possibly to preserve compiled code across executions of the same program [111].

The optimization phases performed by the compiler are shown in Figure 4.6. The compiler uses two levels of internal representation (IR). Bytecodes are loaded and converted from the stack-based representation into the register based high-level bytecode IR. This is followed by a CFG transformation pass that removes dead code, and optimizes branch targets (e.g. a branch targeting a unconditional jump).

Local (intra-basic block) optimizations are performed at this point, primarily to reduce the size of the IR. Local optimizations include copy and constant propagation, common sub-expression elimination and null-check elimination.

Function inlining is performed next. What functions are inlining is determined by a set of function inlining commands provided to the compiler. The inlining commands were determined from a call-edge profile, and modified by hand to improve performance.

1. Convert Bytecodes to High-Level IR
2. Optimize Jump Targets
3. Local Optimizations
4. Function inlining
5. Lazy Code Motion
6. Sparse Copy and Constant Propagation
7. Convert to Low-Level IR
8. Optimize Jump Targets
9. Register Allocation
10. Basic Block Layout
11. Code Generation

Figure 4.6. Strata compiler optimization phases.

There are two primary global optimizations. First, lazy code motion (LCM) [79] is iterated with copy propagation. LCM is a form of *partial redundancy elimination* (PRE) [95] that subsumes global common sub-expression elimination and loop-invariant code motion. LCM is simpler than the original PRE optimization and reduces unnecessary register pressure. The basic LCM transformation is fairly conservative; it only moves code when it can be proven to execute less frequently in the new location. We have modified the algorithm slightly to make it hoist loop-invariant code more aggressively, and to ignore rarely taken paths, such as exception paths. Simple alias analysis is performed along with LCM, which allows redundant loads (such as *getfields*) to be hoisted and combined in most cases. Stores are not eliminated.

Given a chain of dependencies, LCM will only hoist the first operation in the chain. We perform repeated LCM passes to hoist longer chains. Generally, fewer than five passes are needed, and often only two.

Second, a sparse form of copy and constant propagation, integrated with dead-code elimination [131], is performed. The transformation is based on SSA form, so the IR is first converted to SSA form. The algorithm has been extended to eliminate a variety of run-time checks required by the Java bytecodes. These include null reference checks and array bounds checks.

After the IR is converted back from SSA form, it is transformed into a low-level IR. The low-level IR splits many bytecodes into more primitive operations. In some cases this creates small hammocks with inefficient control-flow, so another round of jump optimization is performed.

Register allocation is performed on the low-level IR. Register allocation is performed with a graph-coloring global allocator that also eliminates unnecessary copy instructions [61]. Unneeded copies are produced by the conversion from SSA form, and by marshaling function call parameters.

Two extensions to the algorithm reduce save and restore code around function calls. This is a significant factor when studying speculative object co-location (Chapter 6).

First, the algorithm intelligently uses caller- and callee-saved registers. Variables not live across a call prefer a caller-saved register. Caller-saved registers do not need to be saved and restored by a method's prologue and epilogue, and this leaves more callee-saved registers available. Variables that are live across a method call prefer callee-saved registers.

Second, variables in caller-saved registers are only saved if modified since the previous function call, and only restored if used before the next function call. This is enabled by an additional data-flow pass.

A current limitation of the global register allocator is that it does not support spilling, should the interference graph not be colorable. Almost no spilling occurs, even in the presence of LCM and function inlining, so this is not a serious problem. However, in some cases function inlining commands have been modified to reduce register pressure. If the graph cannot be colored, register allocation falls back on a local register allocator, which produces much inferior code.

While several benchmarks do invoke the local register allocator, those methods do not represent a large fraction of the benchmark execution. Furthermore, we assure that no additional methods fall back to the local register allocator when comparing execution times between different compiler optimizations, such as when exploring object co-location in Chapter 6.

Basic block layout is the final transformation before code generation. Basic block layout orders basic blocks to minimize jumps and taken branches. Our 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. The layout algorithm is a greedy algorithm based on an opportunity cost model, which integrates one level of look-ahead into the optimization.

Measuring Application Progress

The Strata compiler and run-time system provide a number of special features to support research. In particular, the Strata compiler can insert *heartbeats* to provide a deterministic constant measure of application progress, regardless of the optimizations performed or the target architecture. One heartbeat is inserted at the start of every function.

The heartbeat facility enables timing simulation samples to be compared, even between different compilations. Changing the compiler optimizations alters instruction counts, so typical sampling techniques based on instruction counts cannot be used. Samples would represent different portions of the application. Application heartbeats provide a uniform measure of application execution that is invariant across different compilations of the same application. The heartbeat facility has also greatly aided debugging and development, since it provides an invariant measure of when events happen during the execution of a benchmark.

Heartbeats are designed to have minimal impact on compiler optimizations or simulated execution time. A heartbeat node in the IR looks like a function call, but has no inputs or outputs and does not alias with load instructions. As a result, heartbeats do not affect compiler transformations, such as the global code motion performed by LCM.

When producing SPARC code, heartbeats are produced as a call to the run-time system to increment a global counter, resulting in high overhead. However, when producing PISA code, the heartbeats are encoded as an instruction annotation. The simulator notes these annotations during instruction commit, and increments a heartbeat counter within the simulator. Heartbeat annotations do not affect simulator timing. While heartbeats do not usually require any extra instructions, in certain cases an explicit NOP must be added to contain the heartbeat annotation.

In order for heartbeats to be comparable across runs, the application must follow exactly the same function call sequence. The run-time system provides a number of features to make execution more deterministic. Strata uses non-copying garbage collectors, which means that object addresses can be used for object hash codes. However, to do so makes application execution vary slightly, depending on the exact layout of the heap. Therefore, object hash codes are derived by mapping object addresses to unique integers. This is not as efficient as using the object address, but does provide deterministic hash code values at run-time. Another source of variation in executions are accesses to the system clock. The run-time system has the ability to return a deterministic time that increases by 7ms per request.

4.4.2. SimpleMP Simulator

Simulations are performed using SimpleMP, a version of SimpleScalar modified to simulate multiple processors. SimpleMP is completely execution driven, including a detailed MOESI coherence protocol. SimpleMP simulates sequentially consistency. This simulator was extended to support multiprocessing with heterogeneous microarchitectures, SMT and the RPA. The RPA is simulated in detail at the cycle level, including the PCT, the query engine pipeline, message engine, profile buffers and profile networks, as described in Section 4.1.

SimpleScalar PISA has 64-bit instructions that provide 24 extra annotation bits for defining new instructions. This feature has been used extensively. One annotation bit is dedicated to the heartbeat facility described above. Two more annotation bits are used to implement the profile tag described in Chapter 3.

Several support instructions were also added to PISA using annotation bits. Load-locked/store-conditional instructions are used to build synchronization primitives. The SET_PCR and GET_PCR instructions were also added to configure the RPA. Several instructions were also added just for performance analysis. These instructions cause the simulator output execution statistics, and are used to measure the performance and progress of service threads. This feature is used, for instance, when measuring the duration of various phases of GC (see Section 5.4)

4.4.3. Baseline Microarchitecture

The system-on-a-chip we model is shown in Figure 4.7. This model provides a mix of chip multiprocessing (CMP) [99] and fine-grain multithreading (FGMT). On one chip, there is a large high-ILP processor supplemented by three service processors. The computation of greatest concern, the application, runs on the high-ILP processor. Lower priority VM tasks

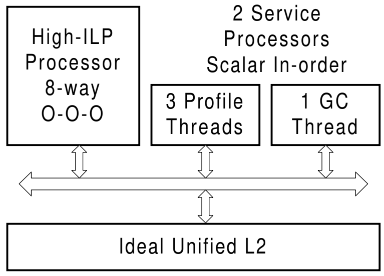


Figure 4.7. System-on-a-chip design.

run on two service processors concurrently with application execution.

All processors execute the same ISA. Different microarchitecture are provided to support the varied requirements of the co-designed system. The high-ILP processor provides high performance for the computation of greatest concern, the application. Service threads can often supply almost unlimited TLP. Therefore the service processors maximize throughput per unit area, rather than single-thread performance at high cost.

Typically one service processor runs the GC algorithm. The second service processor consumes profile records from the RPA using three threads. Two threads process write-barrier records for the GC algorithm (see Chapter 5). The third thread consumes edge-profile records.

Parameters for the processor models are shown in Table 4.3. The ILP processor is a superscalar, out-of-order processor, running only one thread. The out-of-order issue window can contain 128 instructions, of which 64 may be loads and stores. Up to eight instructions can issue in any given cycle, including at most four loads and stores and one multiple or divide. Integer multiplies take 3 cycles. Integer divides take 20 cycles, and are not pipelined. System calls cause a pipeline flush, but O/S code is not simulated. ALU instructions have one cycle of latency. The benchmarks simulated do not execute floating point instructions.

The application processor has a load-to-use latency of three cycles in the case of an L1

Table 4.3. Baseline processor model parameters.

Parameter	*Application Processor*	*Service Processor*	*Units*
Number	1	2	Processors
Threads	1	3	Threads
Width	8	1	Instructions
Instruction Window	128	(in-order)	
Branch Prediction	4KB gshare	Not-taken	
Minimum Penalty	8	4	Cycles
L1 I-Cache	32KB 4-way	1KB 4-way	
L1 D-Cache	64KB 2-way	2KB 4-way	
Load-use-latency	3	2	Cycles
Line Size	32		Bytes
Unified L2	Perfect, shared		
L2 Access Latency	8		Cycles
Shared Bus Occupancy	2		Cycles
Shared Bus Latency	2		Cycles

hit (one cycle to generate the effective address, two cycles to access the cache). The service processors have a smaller, faster L1 D-Cache.

All three processors connect to an ideal L2. A cache miss to the shared L2 adds twelve more cycles of latency—two cycles to send the request, eight cycles through the L2 cache, and two more cycles for the response. Bus contention may increase this number. Cache-to-cache transfers are faster, incurring seven additional cycles of latency.

Service processors are six stage scalar pipelines capable of running three threads using FGMT. To keep these processors small and simple, they have small caches and predict branches to be not taken.

The I-caches are actually modeled as twice the size reported in Table 4.3, because 64-bit PISA instructions are twice the size of a typical instruction set. For normal 32-bit RISC instructions, the caches are nearly equivalent to the size shown in Table 4.3.

Table 4.4. Benchmark description.

Benchmark	*Input*	*Description*
	SPECjvm Benchmarks	
compress	(none)	Lempel-Ziv string compression. Port of the SPECint benchmark
db	100	Simple in-memory database
jack	10	Parser generator
javac	-verbose -classpath input/myclasses.zip JavaLex.java	Java source to bytecode compiler
jess	10	Java Expert System Shell
mpegaudio	10	Audio file decompression
raytrace	20 200 input/time-test.model	3-D ray-tracing rendering
	Other Benchmarks	
espresso	-verbose -obfuscate -d . _209_db/Database.java	Java source to bytecode compiler.
java_cup	< java_cup/simple_calc/parser.cup	Parser generator
jlex	sample.lex	Tokenizer generator
pizza	-verbose src/examples/mergesort.pizza	Pizza language compiler
rpa	swbr256_prof.prx < swbr256_prof.pas	RPA assembler
strata	-arch ss -allopts -gc_genMPSS -uvar -lcm -lcmSpec -ver 1 spec.benchmarks._209_db.Database	Strata VM bytecode compiler
xp	xml/conf/japanese/pr-xml-utf-8.xml	XML parser

4.4.4. Benchmarks

The 14 benchmarks shown in Table 4.4 are simulated. Seven are taken from the SPECjvm98 benchmark suite. *strata* is Strata's bytecode compiler. *rpa* is the assembler for the RPA assembly language. The other six applications are freely available language tools.

4.4.5. Benchmark Measurement Intervals

Many of the benchmarks are short enough to be run to completion. When this is not the case, a specific measurement interval is simulated to provide an accurate view of the benchmarks overall performance characteristics and minimize transient initialization effects while keeping simulation times reasonable. The intervals described here were developed to study the garbage collector in the next chapter. They are used here also for consistency.

One large interval of about 400 million instructions is simulated near the beginning of benchmark execution. Multiple smaller intervals could also be simulated, but these are unlikely to capture complete garbage collections. For all benchmarks, VM initialization and application start-up code is executed functionally, typically to a point a few million cycles before the first GC. This is followed by a short warm-up period to warm the processor caches, followed by the measured interval.

Table 4.5 indicates the size of the intervals measured. The warm-up period and interval are specified in terms of application heartbeats so they are consistent across different executions of the same benchmark. Table 4.6 shows the execution time of each interval. The cycle counts are taken while running a base-line configuration. Store instructions are profiled to support the concurrent garbage collector, and one in 256 branches are profiled to simulate edge profiling. The profiler has four profile networks and eight profile buffers. Note that several of the benchmarks, notably *rpa* and *java_cup* are quite short.

4.4.6. Simulated Queries

The RPA is driven using two example queries. These are edge profiling and concurrent GC, as shown in Figure 3.3. Edge profiling is a simple, common profiling task, and the behavior of the RPA can observed under different loads by varying the random sampling rate. The GC write-barrier query samples all executions of certain store instructions. This

Table 4.5. Measurement interval heartbeats (thousands).

Benchmark	*Fast-Forward* (1)	*Warm-Up* (2)	*Interval* (3)
compress	615	85	20000
db	10	10	5230
espresso	97	3	Completion
jack	800	15	3335
java_cup	81	19	Completion
javac	10	10	5780
jess	84	16	Completion
jlex	80	20	Completion
mpegaudio	1560	40	2700
pizza	75	25	Completion
raytrace	75	25	Completion
rpa	5	5	Completion
strata	80	20	3335
xp	60	40	Completion

Table 4.6. Measurement interval sizes (millions).

Benchmark	*Instructions (Millions)*			*Cycles (Millions)*		*Interval*
	Fast-Forward (1)	*Warm-Up* (2)	*Interval* (3)	*Warm-Up* (4)	*Interval* (5)	*IPC* (6)
compress	31.8	1.7	400.7	0.8	151.5	2.64
db	10.0	4.0	456.5	2.0	161.4	2.83
espresso	35.8	2.0	43.7	0.5	16.1	2.71
jack	120.1	1.2	398.9	0.6	134.7	2.96
java_cup	24.3	1.5	26.3	0.7	10.1	2.60
javac	10.7	3.8	419.9	1.6	150.7	2.79
jess	30.6	1.8	228.1	0.5	80.3	2.84
jlex	21.9	1.8	52.8	0.6	16.1	3.28
mpegaudio	247.8	5.6	401.9	1.3	88.4	4.55
pizza	26.8	1.8	17.5	0.7	8.5	2.06
raytrace	21.0	1.6	213.7	0.6	82.1	2.60
rpa	8.8	1.9	7.9	0.8	4.1	1.93
strata	32.6	3.2	392.8	1.2	144.0	2.73
xp	17.0	2.0	32.7	0.5	11.7	2.79

is a demanding instrumentation task, because every such store must be sampled.

4.5. Performance Analysis of the RPA

The section describes simulations used to evaluate the performance impact of using the RPA, the amount of hardware resources the RPA needs, and the effect on system parameters such as bus bandwidth utilization. Six of the 14 benchmarks, *compress*, *jess*, *jlex*, *mpegaudio*, *raytrace* and *rpa*, place less stress on the RPA because they rarely execute instrumented store instructions. Results focus on the other eight benchmarks, *db*, *espresso*, *jack*, *java_cup*, *javac*, *pizza*, *strata* and *xp*.

4.5.1. Resource Usage

The initial goal is to establish reasonable hardware configuration parameters for the RPA. Simulations determine the number of profile buffers and networks the RPA needs for low-overhead operation.

Figure 4.8 and Figure 4.9 show results from simulating the edge profiling and GC queries from Figure 3.3. One out of 256 branches are sampled. Profile resources are essentially unrestricted with 32 profile buffers and 32 profile networks.

Figure 4.8 shows the percentage of cycles (y-axis) in which x profile buffers were in use (x-axis). Eight profile buffers, 256 bytes of storage, appear to be sufficient, except for *jack*. *jack* uses as many as twelve buffers. Further simulations shows that eight buffers will provide moderate overhead even for *jack*.

The behavior of *jack* is unique, and deserves special attention. *jack*'s performance can be traced to two unique application characteristics. First, *jack* writes object references much more frequently than the other benchmarks. Second, the written references are almost all null. Hence, while *jack* profiles many instructions, it produces few RPA messages, since the

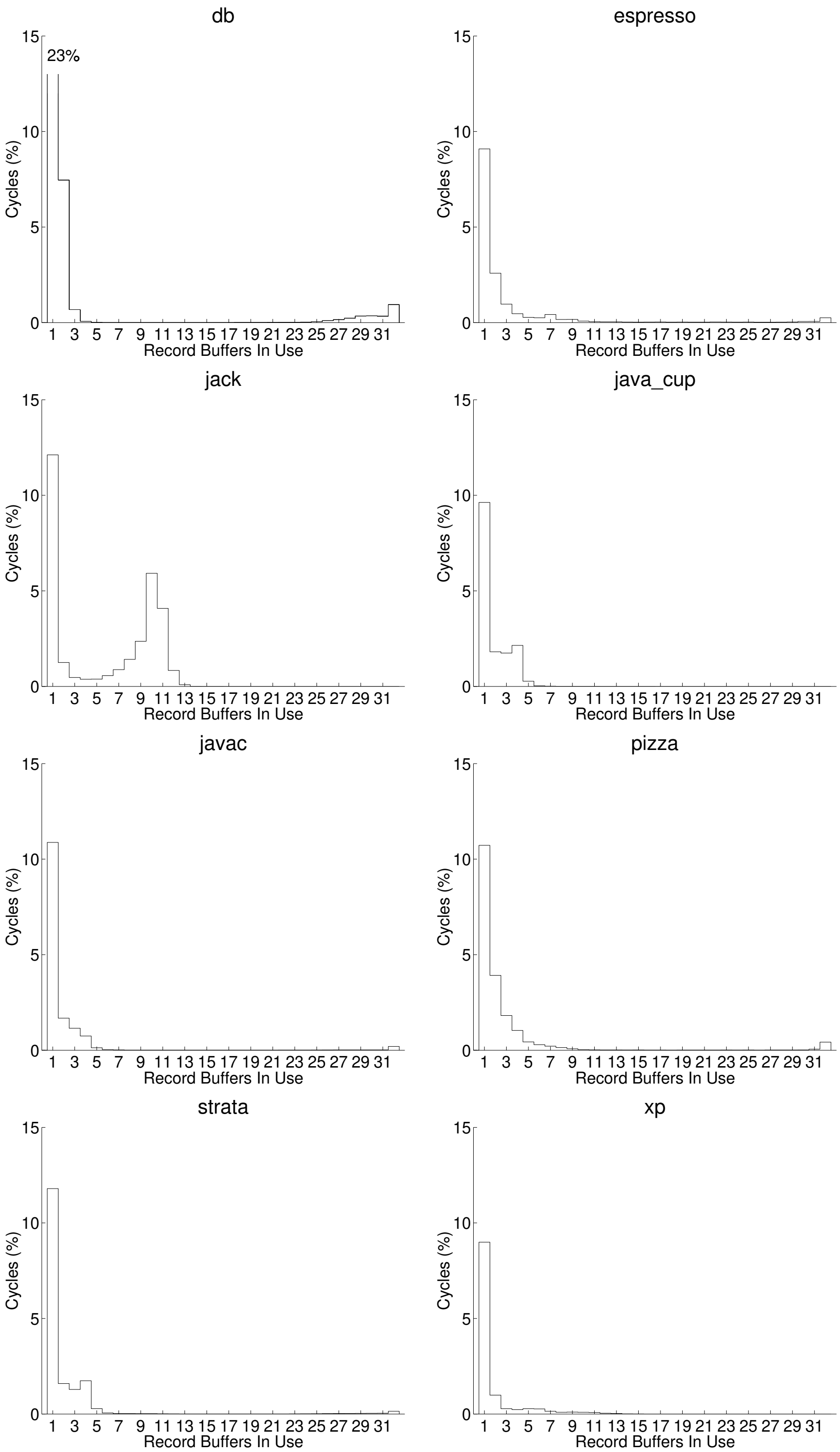


Figure 4.8. Histogram of profile buffers occupied.

GC write-barrier query of Figure 3.3c filters them out.

A single store instruction associated with the standard Java library, *java.util.HashtableEnumerator.hasMoreElements()*, produces this behavior. This function scans a hash table to see if it contains more elements. The function unnecessarily updates an internal reference after scanning each entry in the hash table. It is this update that is producing so many null-reference writes. Although *jack* never inserts more than three elements into the hash table, it allocates tables using the default size of 101 elements. As a result, iterating over the table produces bursts of null reference writes.

A number of the benchmarks show small bumps at the right hand of the histogram, indicating all 32 profile buffers are occupied. This is most notable for *db*, but also occurs on *espresso*, *javac*, *pizza* and *strata*. This phenomenon occurs when the service threads processing the profile records fall behind. The message queues fill, followed by the 32 profile buffers, eventually stalling the pipeline. This appears to happen only transiently, perhaps during bursts of profiling activity. The number of service threads can be increased, and this is investigated more thoroughly while evaluating the garbage collector in Section 5.4.

Figure 4.9 shows the number of profiled instructions simultaneously in flight. The histograms show the percentage of cycles (y-axis) in which x profiled instructions were in flight (x-axis). This includes profiled instructions between the dispatch and retire stages. Under the interconnect model described in Section 4.2, one profile network is need for each in-flight instruction. The histograms are remarkably similar in shape to those of Figure 4.8. They are slightly lower because profile buffers remain allocated longer than the flight time of an instruction. All benchmarks run well with four profile network except *jack* which appears to need as many as ten networks. Direct measurement of profiling overhead in the next section shows four networks are sufficient.

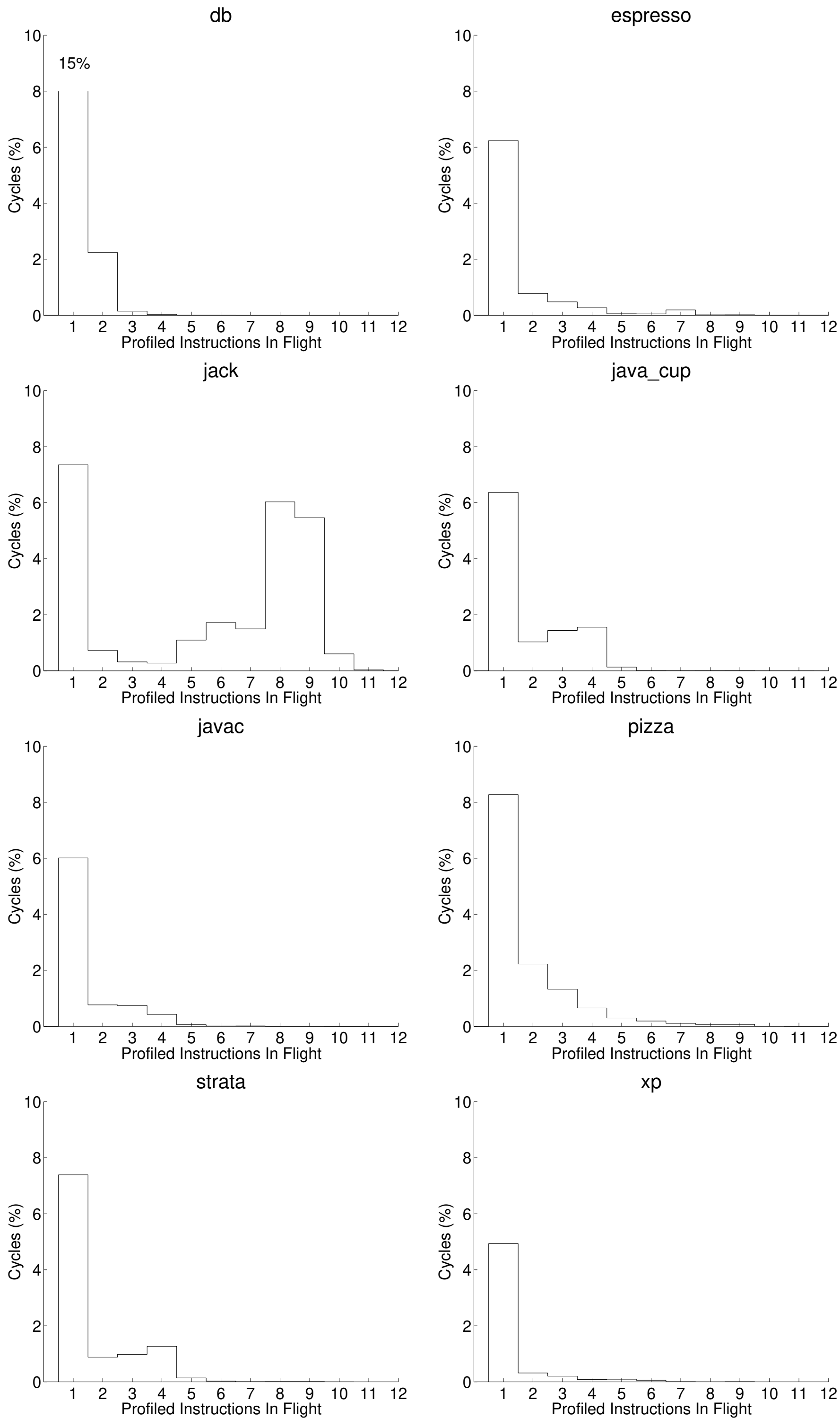


Figure 4.9. Histogram of in-flight profiled instructions.

4.5.2. Performance Overhead

Based on the previous results, the number of profile buffers is reduced to eight, and the number of profile networks is varied from one to six. Figure 4.10 plots the percentage of cycles for which dispatch stalled due to limited RPA resources. Dispatch stalls do not directly reduce performance. The out-of-order execution window covers many of the dispatch stalls.

Most of the benchmarks perform well using four profile networks. The two profiling applications typically profile no more than four instructions at a time. The *jack* benchmark is again an exception to this rule. Dispatch stalls drop from about 5% to less than 2% by adding one more profile network. Nevertheless, we choose to use four profile networks as our baseline model. Improving one of 14 benchmarks by a few percent does not justify the additional resources.

Figure 4.11 plots the percent slow-down resulting from these stalls. These results are obtained by comparing the execution time with many profiling resources (32 profile buffers and networks as in Figure 4.8 and Figure 4.9) to the execution time with eight profile buffers and one to six profile networks. This will yield the correct overhead due to profiling only if all stalls are eliminated with many resources. For those benchmarks in which the service threads fell behind, this is not the case.

This leads to a small amount of error in computing the profiling overhead. Effected benchmarks are plotted using error bars. The number of dispatch stalls with many profiling resources is directly measured. Execution time is increased by some fraction of these stalls.

Not surprisingly, *jack* has the largest slow-down of 3.8% when given four profile networks. All other benchmarks have an overhead of less than 0.6%.

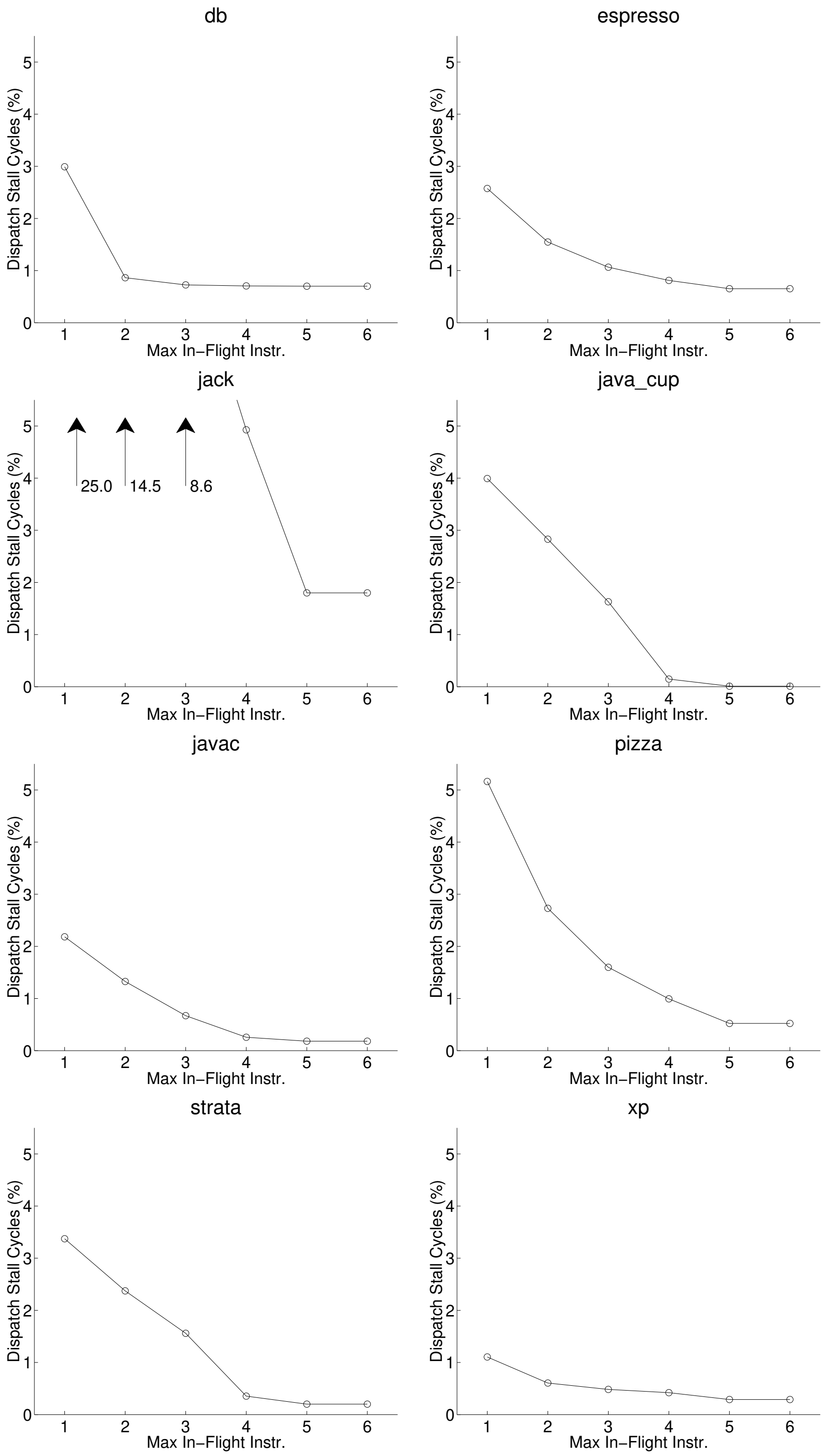


Figure 4.10. Dispatch stalls vs. available profile networks.

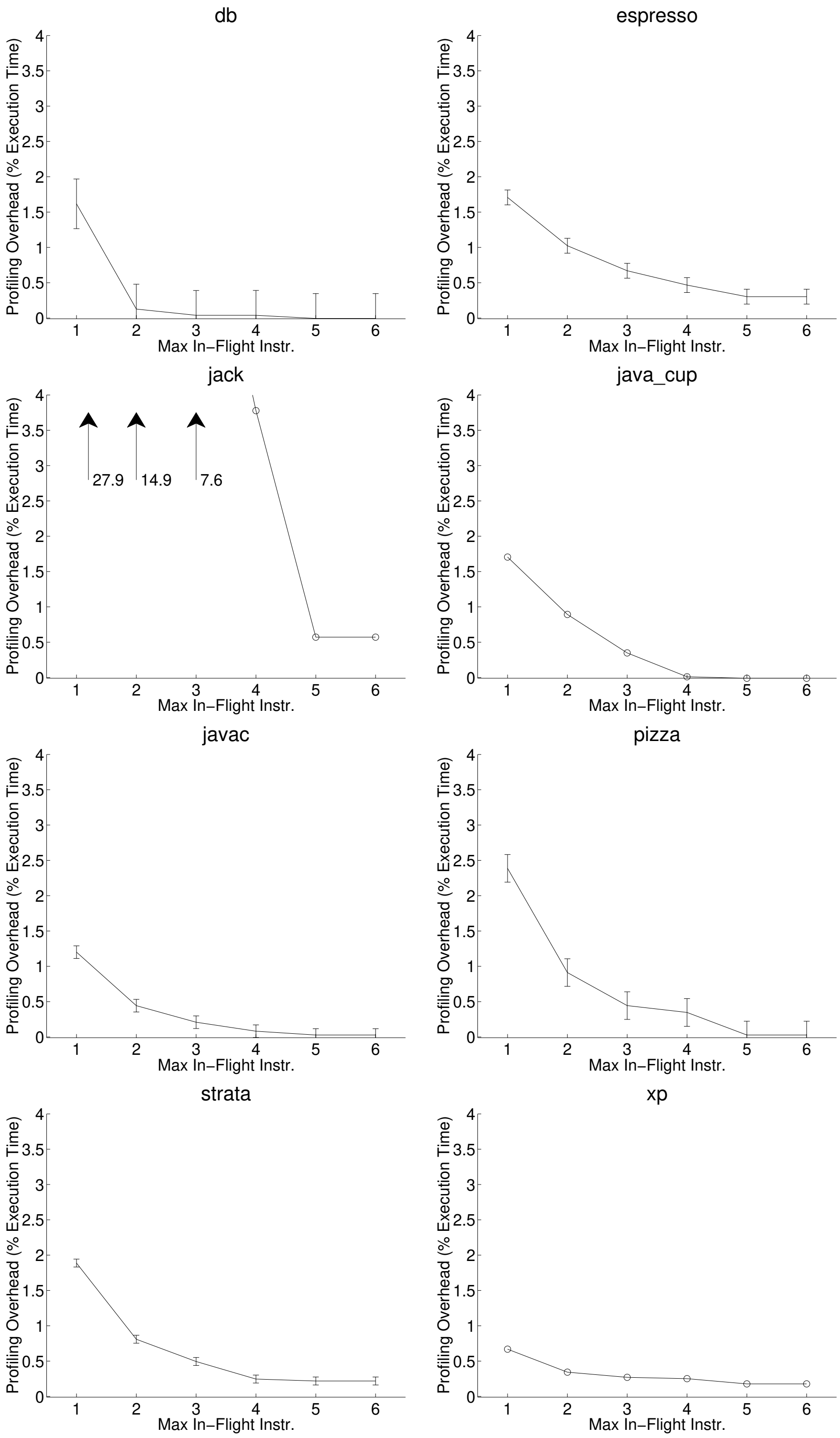


Figure 4.11. Profiling overhead vs. available profile networks.

4.5.3. Profiling Rate

To understand how profiling overhead varies with the sample rate, four sampling rates for edge profiling are simulated; rates simulated are one in 32, 64, 128 and 256 branches. Note that the GC query is still included. Figure 4.12 plots the slow-downs for these rates. All benchmarks show a sharp increase in overhead between sampling one in 64 branches and one in 32 branches. When sampling one in 64 branches, overheads are still less than 1% except for *jack*(4.8%), *xp* (1.6%) and *pizza*(1.0% $\pm$ 0.2%).

Table 4.7 shows the estimated sampling rates that this system can handle. This table shows the profiling rates estimated to yield a 2% performance overhead. Sampling rates are given in instructions per sample and cycles per sample for each benchmark. These values are generated by linear interpolation between the branch profiling rate just above 2% overhead and just below 2% overhead. In all cases these are the 1:32 and 1:64 data points, with the exception of *jack*, whose overhead is always above 2%. For *jack*, the instruction profiling rate is shown when profiling one branch in 256. Although the high profiling rate makes *jack*'s high overhead understandable, its overhead is actually quite *low* for its profiling rate. This is likely because most of the profiles are filtered by the query engine, as is discussed in Section 5.4.

The results show that the RPA can profile one instruction every 46 (*db*) to 84 (*java_cup*) cycles, and one instruction every 128 (*db*) to 225 (*xp*) instructions.

4.5.4. Bus Bandwidth

In order to be practical, the bus bandwidth used by the RPA should be low. The modeled system-on-a-chip contains a shared bus connecting all three processors to the shared L2. Table 4.8 shows the percentage of the shared bus used by RPA messages. Although traffic between the L2 and memory is not modeled, most profile records should not need to go to

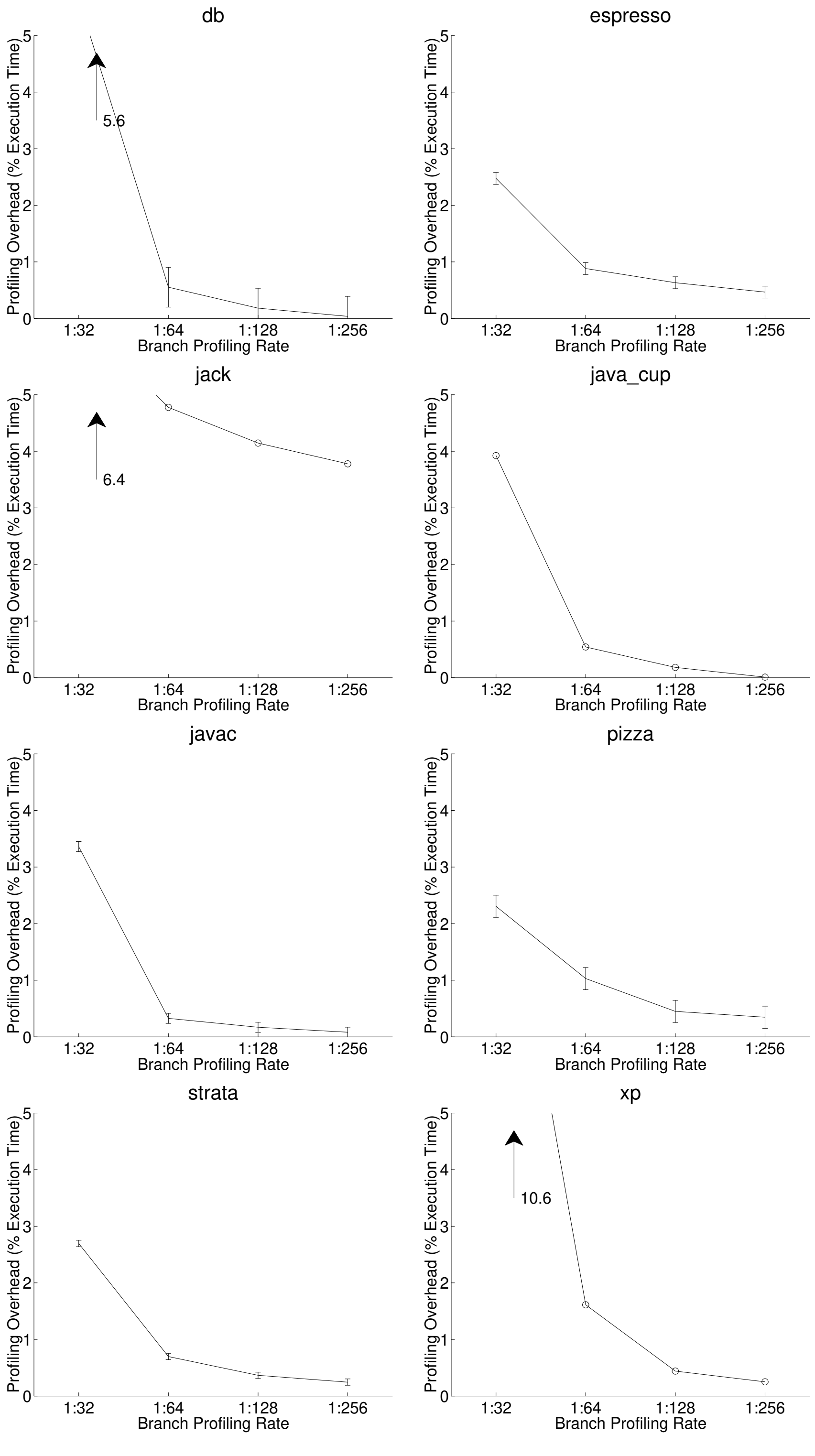


Figure 4.12. Profiling overhead vs. branch sampling rate.

Table 4.7. Estimated sampling rates yielding 2% profiling overhead.

Benchmark	*Cycles per Sample (1)*	*Instructions per Sample (2)*
db	46	128
espresso	62	165
jack	> 14	> 41
java_cup	84	215
javac	61	169
pizza	64	129
strata	73	196
xp	82	225

memory. The message queues, totaling 3KB, will easily remain cached within the L2.

The shared bus contains an address bus, used for requests and acknowledgments, and a data bus used to transfer cache line data. Each bus has an occupancy of two cycles. The address bus has dedicated wires for requests and acknowledgments; acknowledgments do not require additional bandwidth. While point-to-point networks are expected to be common on future CMPs, the design studied is simple and likely conservative. In particular, single-cycle occupancy would double the bus bandwidth, halving the utilization shown in Table 4.8.

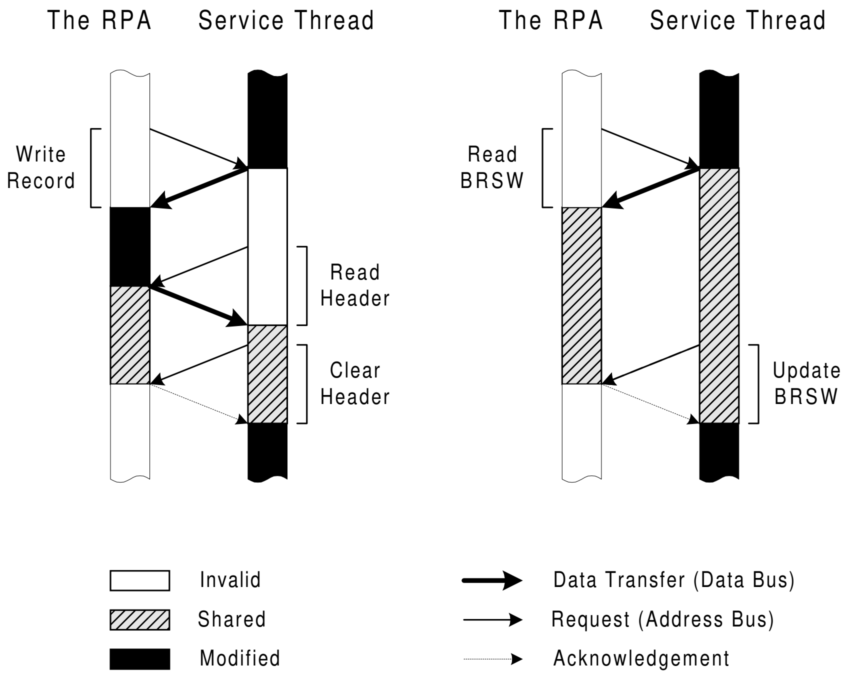
The results are tabulated while profiling one out of 256 branches. For all benchmarks except *db*, the address bus utilization is less than 5%. For *db* it is only 8.7%. Data bus utilization is slightly lower.

These numbers are derived from the measured cycles per message, and a detailed understanding of the RPA and the coherence traffic generated by the MOESI protocol. This interaction is diagramed in Figure 4.5.4. Figure 4.5.4a shows the bus transactions that occur when transferring a message. Figure 4.5.4b shows the transactions that occur when polling the buffer read status word.

When the RPA writes a new record, it is guaranteed to miss, because the service thread must have cleared the header, reclaiming the line from the RPA processor. To write the line,

Table 4.8. Estimated bus bandwidth used by RPA messaging.

Benchmark	*Cycles Per Message (1)*	*Percentage of bus bandwidth used.* Address Bus (2)	Data Bus (3)
db	74.7	8.7	6.7
espresso	135.4	4.8	3.7
jack	180.6	3.6	2.8
java_cup	191.2	3.4	2.6
javac	166.7	3.9	3.0
pizza	151.2	4.3	3.3
strata	166.7	3.9	3.0
xp	175.7	3.7	2.8



a) Message Transfer **b)** Buffer Read Status Word (BRSW)

Figure 4.13. MOESI protocol transactions associated with RPA messages.

the RPA issues a bus command (read-for-ownership), which retrieves an exclusive copy of the line. When the service thread reads the message, it first polls the header word, resulting in a bus command (read), which retrieves a shared copy of the line. When the service thread is finished with the line, it must clear the header word. Unfortunately it only has a shared copy, so a second bus command (upgrade) invalidates the RPA's copy. However, no data response is needed, since the service thread has the line in its L1 cache. Thus two data bus transactions, and three address bus transactions are generated per message.

Under heavy load, the algorithm used by service threads to consume messages will update the buffer read status word once every four messages. See Subsection 3.2.1. The RPA polls the BRS word, periodically. How often depends on how full the message queue is. However, in the worst case the RPA will access the BRS word every time it is updated by the service thread. This is the scenario shown in Figure 4.5.4b. The service thread has just updated the value of the BRS, and the RPA is reading the new value. This requires an address bus transaction (read) to retrieve a shared copy of the line. After reading another four messages, the service thread updates the BRS. This requires a bus command (upgrade), but does not require a data transfer, assuming the BRS word is still in a shared state in the service thread's cache. The result is two address bus transactions and one data bus transaction for *every four* messages.

In total, 3.25 address bus transactions and 2.5 data bus transactions are generated for each message sent.

4.5.5. Profile Network Latency Tolerance

Figure 4.14 demonstrates the latency tolerance of the profile network. Profile records are not made available to the query engine immediately after the profiled instruction retires. Instead, to simulate profile network latency the record is not made available for some number

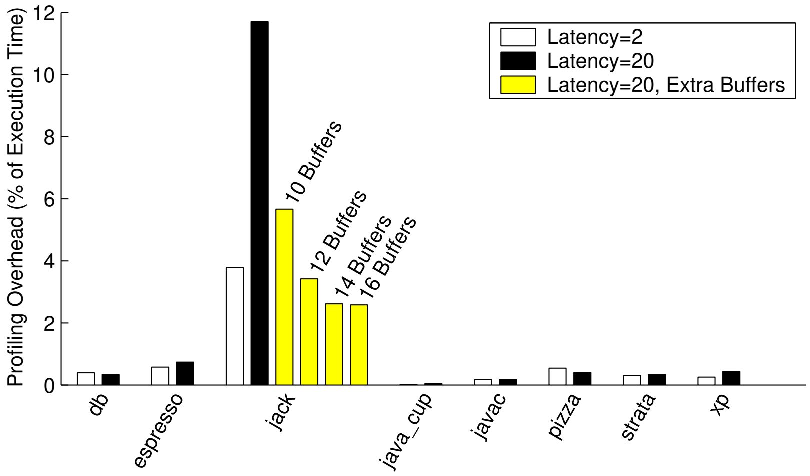


Figure 4.14. Latency tolerance of the profile network.

of cycles after the instruction retires.

Previous results assume a two cycle network latency. Figure 4.14 demonstrates the effects of increasing the network latency to 20 cycles. An edge sampling rate of one in 256 branches is used. The figure plots the slow-down while using the two-cycle network latency, and slow-down from a 20-cycle network. In both cases, the slow down is compared to a profiler with 2-cycle latency, and 32 buffers and profile networks. For all benchmarks, except *jack*, there is little change.

The additional latency dramatically increases the overhead of *jack*. *jack* demonstrates that additional profile buffers can be used to compensate for network latency. Longer network latency causes the profile buffers to remain occupied longer, since it increases the time between when buffers are allocated at instruction dispatch, until they are freed when the query completes. The detrimental effects of holding buffers longer can be countered by providing more buffers. Several additional runs were simulated for Jack, with 10, 12, 14 and 16 buffers. As can been seen, in Figure 4.14, four extra buffers more than compensates for the increase in network latency.

4.6. Related Work

Special-purpose mechanisms have been proposed for many profiling tasks. These require special hardware, software, or both. Conte et al. propose two methods [38, 37] for edge profiling. The first samples the values of the branch-target-buffer and branch prediction array to derive an edge profile. The second method improves the accuracy using a small special-purpose array, the *Profile Buffer*. The Profile Buffer is indexed by branch PC, and caches counts of taken and not-taken branch executions. This buffer is updated only for committed branches when they retire. In both methods the tables are periodically read by software using interrupts. The RPA provides this functionality by sampling branches (see Figure 3.3a), and building the Profile Buffer in software using a service thread.

Merten et al. [90] develop a scheme for identifying hot spots in programs. Their scheme works by collecting branch taken/not-taken counts in the Branch Behavior Buffer (BBB), a structure similar to the Profile Buffer, though much larger. The BBB also identifies frequently seen branches and uses this information to identify hot spots. A separate structure, the Monitor Table prevents hot spot re-detection. In the context of this work, the RPA would profile branches, and service threads would perform the hot-spot detection algorithm.

Several elements of the ProfileMe mechanism [41] also appear in the RPA. ProfileMe provides a simple form of instruction-based profiling. The hardware picks one instruction from the stream, using a software-settable decrementing counter. Simple random sampling is performed by resetting the counter to randomized values. ProfileMe allows information to be collected for both retiring and squashed instructions. When the instruction is squashed or committed, an interrupt allows software to read a fixed record of information. To keep costs low, ProfileMe keeps sampling rates low, e.g. one sample every 10^3 to 10^5 instructions, much lower than the one in 150 instructions sampled by the RPA.

ProfileMe also supports paired sampling, a simple and powerful form of clustered sampling. Paired sampling collects two instruction samples close to each other using a major and minor sampling interval. Paired sampling is useful for measuring interactions between instructions. The RPA is not incompatible with paired sampling; adding paired sampling to the RPA presents an interesting and likely useful research challenge.

A mechanism described in a patent by Westcott and White [132] is also very similar to both the RPA and ProfileMe. The processor uses a counter to randomly select instructions for profiling. Like ProfileMe, a standard profile record is collected. Like the RPA's query engine, *triggers* can be used to scan for records of interest. Also like the RPA, the records are stored to a buffer in memory. However, the buffer is read following an interrupt when the buffer fills.

Contemporaneous with this work, Zilles and Sohi developed a profiling mechanism with many similarities to the RPA [39]. It selects instructions for profiling based on their op-code class and the low-order bits of the instruction PC using a hardware filter similar to the PCT. The mechanism divides PCs into buckets, and can control sampling on a per-bucket basis. Collected information is held in a retired instruction buffer (RIB) (like RPA's profile buffers), until examined by a simple programmable profile co-processor. The mechanism collects records for as many instructions as it can; if the RIB fills, records are dropped. This co-processor summarizes information before it is relayed to software via an interrupt. The co-processor is a more complete processor than the query engine, having registers, a private data memory and an associative array for hash tables. The RPA off-loads much of this computation to service threads.

Sastry et al. evaluate *stratified sampling* as extension to a wide range of hardware-assisted profilers, including the RPA [108]. Stratified sampling reduces the bandwidth of the profile stream, or increases the accuracy with a given number of samples. A hashing function subdivides the incoming profile stream into many separate substreams. Each substream will tend to be highly *biased*, dominated by one or a few common records that hash into that substream. It is well known that more highly biased populations can be estimated more accurately.

The profiles of the individual substreams can be precisely recombined if the size of each substream is maintained exactly, or nearly exactly. The result is higher overall accuracy. Stratified sampling can also be viewed as a compressing the profile stream by collapsing identical profile records together.

The RPA has some additional capabilities not present in previous mechanisms. Working in the context of service threads leads the RPA to focus on communicating profiled information through shared memory. All previous profile mechanisms use interrupts to record or sample profiled information. The RPA can also emulate instrumentation. It will guarantee information from some instructions is always collected. This allows the profiler to be used for new applications that require profile information for correctness.

4.7. Summary

The relational profiling architecture (RPA) provides a powerful and flexible profiling mechanism. It enables the same optimizations as several previous mechanisms, and has additional capabilities as well. Because the RPA can emulate instrumentation, it can be used even when the information is required for correctness. Working in the context of service threads leads to communicating profile information through shared-memory queues, which increases the sampling rate and lowers profiling overhead, compared to previous interrupt-driven approaches. This leads to new applications for hardware-assisted profilers such as concurrent garbage collection.

Simulation results shed light on the implementation requirements and profiling overhead of the RPA. Profiling four simultaneous in-flight instructions and storing eight profile records is sufficient to make losses negligible. The profile interconnection network, which scales linearly with the number of simultaneously profiled instructions, appears to be the greatest cost of profiling. The PCT reduces this cost by selectively profiling only the particular data that is needed. This reduces the size of the profile records, and could also be used to intelligently allocate the profile network. The profile network is latency tolerant, and additional profile buffers can compensate for long latencies.

The PCT allows the sampling rate to be tuned to match the frequency with which the profiled event occurs. For common events, low sampling rates reduce the bandwidth required. For rare events, high sampling rates reduce the time required to obtain a representative sample. In addition, different benchmarks stress the profiling mechanism to different degrees. This suggests that the RPA should monitor its own behavior so as to adapt the sampling rate to the application.

In summary, the RPA provides a good foundation for the development of profiling tasks. In the next two chapters, two such tasks are developed in detail. These are the concurrent garbage collector, also simulated above, and object co-location.

CHAPTER 5

CONCURRENT GARBAGE COLLECTION USING THE RPA

Garbage collection (GC), the automatic reclamation of memory no longer needed by the application, is commonly required by object-oriented languages such as Java. Although, GC algorithms have been researched for decades, it still exacts a significant performance penalty unless memory is used inefficiently [10, 100, 145].

Performing garbage collection concurrently with application execution overlaps some of the work. However, such collectors usually have *more* performance overhead than non-concurrent collectors [14, 71, 83, 100], due to the cost of synchronization between the collector and the application.

Concurrent GC provides an ideal opportunity to apply the RPA to a non-traditional profiling task. Concurrent GC algorithms use *barriers* to monitor certain application activities, typically reads and writes of object references. The RPA provides the primitives needed to do this efficiently.

The RPA instruments store instructions selected by the VM, stores that write references into objects. The reference value written and the object modified, available as the input operands to the store, are collected and passed to service threads. The service threads perform the *write barrier* on behalf of the collector, itself running as another service thread. This eliminates the overhead associated with software inlined write barriers, and replaces previously proposed special-purpose hardware [109, 138, 93] with a general profiler useful for a wide range of tasks.

Beyond reducing overhead, the RPA makes the write barrier more flexible. In this algorithm, the behavior of the store barrier changes, depending on the phase of collection. Modifying the behavior of inlined write barriers is not practical. However, the RPA and service threads provide this type of flexibility. The service threads just check a global variable to determine what type of write barrier to execute. This check is only performed when the write-barrier threads have no records to process, so it does not reduce performance.

The RPA also reduces the work done by the write-barrier threads. Since the collector does not need to observe null-reference stores, the query engine discards these profile records. This simplifies the write-barrier code, and reduces the number of profile records that must be processed. For the *jack* benchmark this eliminates 95% of the write-barrier messages.

The result is an extremely efficient concurrent generational garbage collection system. On average, the benchmarks spend 0.4% of their time performing GC. Overhead comes from two sources. First, the application stalls which profiling resources are exhausted, as discussed in Section 4.5. Second, the application must be paused for three short periods during collection. These pause times are short, at most 55 thousand cycles.

We continue with a discussion of concurrent garbage collection algorithms, including prior hardware support Subsection 5.1.3. Section 5.2 describes the algorithm developed here, omitting aspects related to generational collection for simplicity. Section 5.3 describes the generational extensions to the algorithm. Section 5.4 presents simulation results evaluating the behavior of the algorithm. Section 5.5 summarizes and concludes the chapter.

5.1. Concurrent Garbage Collection

Concurrent GC has a long history. Developing an algorithm that is both efficient and correct has proven to be difficult. This section provides an overview of algorithmic issues, previously implemented algorithms and previous hardware support for GC. GC algorithms are generally divided into reference counting algorithms and mark-sweep algorithms. This review focuses on mark-sweep algorithms.

5.1.1. Concurrent Mark-Sweep GC

Mark-sweep GC algorithms find live objects by recursively traversing references from previously found objects, marking objects along the way. Marking begins with a set of *roots*, objects directly referenced by global or local variables. At the end of the marking phase, all unmarked objects known to be unreachable by the application and can be collected. Sweeping scans the heap for unmarked objects and collects them.

Concurrent GC algorithms must deal with concurrent modifications of the heap. *Barriers* solve this problem by observing certain application activities, typically reads and writes of references.

Sweeping, compared to marking, is easy to perform concurrently with the application because sweeping only touches dead objects. Since dead objects are inaccessible by the application synchronization problems are few. Consequently, most literature focuses on the more difficult problem of marking, as we will here.

Some of these algorithms were developed as incremental algorithms to reduce pause times on uniprocessors. Usually they could be made completely concurrent, were multiple processors provided.

5.1.2. Barriers

Figure 5.1 illustrates the difficulty of concurrent marking with a simple example based on the common three-color analogy [47]. In Step 1, GC has marked and scanned object A, placing object B on the mark stack. Before the GC scans object B, the application moves the reference from B to C into object A (Steps 2 and 3). Finally in Step 4, GC scans object B. Because a reference was concurrently moved, GC mistakenly collects object C.

For correctness, scanned objects (black) should always be separated from unreached objects (white) by a frontier of reached but unscanned objects (gray). The example violates

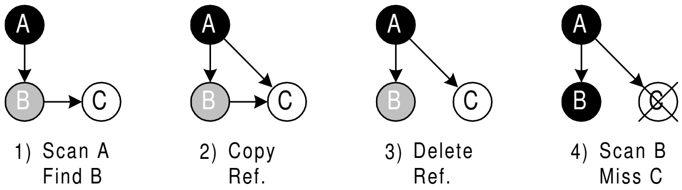


Figure 5.1. Example concurrent reference mutation.

this invariant by moving a reference across the gray frontier, establishing a direct connection from a black to a white object.

This can occur whenever an application moves a reference from one location to another. Moving a reference involves first making a copy of it (Step 2), and, second, overwriting the original (Step 3). Neither copying nor overwriting alone is sufficient to undermine the GC process. Copying alone is not a problem; the original reference still exists to be traced during marking. Overwriting alone is not a problem; if the reference is actually gone, it need not be followed.

Hence, corrective action must be taken when a reference is moved. The process of moving a reference can be monitored by barriers in at least four ways. Referring to Figure 5.1:

Solution 1. In Step 2, the load of the B-C reference is detected, graying C. This requires a read barrier to examine loaded results.

Solution 2. In Step 2, the store of the A-C reference is detected, graying C. This requires a write barrier to examine stored values.

Solution 3. In Step 2, the store of the A-C reference is detected, graying A. This requires a write barrier to examine modified addresses.

Solution 4. In Step 3, the over-write of the B-C reference is detected, graying C. This requires a write barrier to examine overwritten values.

There are several possible optimizations and variants for each of these. In solutions 1, 3 and 4, it is possible to scan the critical object(s) immediately, turning them black instead of gray. Also, the unit of marking may not be an object, but some other convenient segment, such as a virtual memory page.

Wilson [134] divides these solutions into read-barrier solutions (Solution 1) and write-barrier solutions (Solutions 2-4). Write-barrier solutions are further subdivided into snap-shot-beginning (4) and incremental update solutions (2 and 3). [1]

Reference reads are typically much more common than reference writes, so write barriers are generally preferred over read barriers. This is a disadvantage of solution 1. Nevertheless, solution 1 is used by derivatives of Baker's popular algorithm [15] for reasons that will become clear in the next subsection.

Solution 3 has the disadvantage that concurrent modifications cause the collector to back-up (graying black objects). Guaranteeing forward progress is an issue with algorithms using this solution. However in some cases, such as when page protection is used as described below, it is easier to extract the modified address than the stored value. In addition, the modified address is often needed anyway for generational collection. The RPA can easily provide both the stored value and the modified address.

Solution 4 requires the overwritten value, which is not naturally available in the instruction stream. Furthermore, in a multithreaded environment this value represents shared data. Synchronization of some kind is required for most consistency models.

The above considerations lead us to choose solution 2 for our algorithm. Only writes need a barrier, only information naturally available during the write is required, and forward progress is guaranteed.

Copying objects concurrently creates additional complexities. Baker's incremental copying algorithm [15] addresses these issues, and is commonly used as a basis for copying algorithms. The heap is divided into a from-space and to-space. From-space contains the

[1] Wilson also refers to a variant of solution 1 as a snap-shot-beginning barrier.

objects to be collected. To-space is initially empty. The collector copies all live objects into to-space. From-space then contains only dead objects and is reclaimed en masse.

While an object is being copied, the application must see a single consistent copy. Also, while references to the object are being updated to the new location, the application must see only a single value. This is important for reference comparisons in the typical case where the object's address is used as the reference.

Baker's algorithm solves both problems by maintaining an important invariant. The application may only obtain references to objects in to-space. A read barrier accomplishes this feat by examining all loaded references. Whenever the application attempts to read a reference to an object in from-space, the barrier redirects the reference to the copy of the object in to-space, copying the object first if necessary. Thus, this algorithm uses solution 1 above to handle both copy consistency and concurrent heap modifications.

Assuming from-space is a contiguous region, the RPA can perform this test by doing a range check on loaded reference values, and throwing a synchronous exception. However, the frequency of such loads is liable to overwhelm the profile mechanism. Also, unlike most barrier mechanisms, a Baker-like read barrier is synchronous; the barrier must be performed before the application can continue. The amount of worked performed by the barrier may be relatively large, including copying objects and updating references in the heap. For these reasons algorithms based on Baker's algorithm were avoided in this research.

5.1.3. Previous Hardware Support

Straightforward implementations of barriers use inlined barrier code around appropriate loads and stores in the application. However, many hardware mechanisms have been proposed to support specific GC algorithms and particular languages. Such specific hardware mechanisms have rarely been popular with commercial hardware vendors. One goal of our

RPA-based approach is to be general enough to be useful for a wide variety of tasks and algorithms, and still provide the performance improvements of dedicated hardware.

An early but very impressive GC system based on Baker's algorithm was developed for the Symbolics 3600 [93]. This early example of a co-designed VM was designed explicitly for LISP. The GC algorithm is a concurrent copying generational GC algorithm. Tagged memory distinguishes references from other data for both hardware and software. The read barrier is implemented by special hardware that checks the result of every load instruction. Loading a reference to old-space generates an interrupt. The trap handler redirects the loaded word to the copy in copy-space, copying the object first if necessary.

The Symbolics 3600 also implements the generational write barrier in hardware. Page marking [134] is used to track inter-generational pointers. When a reference to a young object (called ephemeral in [93]) is stored, hardware sets a special bit associated with the modified page. These pages are examined for references when the young generation is collected.

Schmidt and Nilsen [109] propose adding hardware support for Baker's algorithm [15] to the memory modules, rather than the CPU. They suggest that adding specialized support in a standard memory expansion slot will be economically more attractive than modifying the CPU. The garbage-collected memory module (GCMM) contains a to-space and from-space, and performs the collection algorithm. Tagged memory identifies pointers for the GCMM. Since caches are typically placed between the processor and the GCMM, the GCMM sees reads and writes of whole cache lines. The GCMM implements the read barrier by trapping all cache-line reads, so the read barrier only executes on cache misses. The trap handler redirects into to-space any references it contains to from-space, ensuring that the application never obtains references to from-space. Copying of objects does not occur at this time. Rather, space is merely reserved so that the copy can be done lazily. Caches must be flushed before collection to ensure that the read barrier processes all references.

The MUSHROOM system [138] used novel forms of support to enable incremental collection of main-memory and in-cache collection of the youngest generation. The young generation is maintained in the cache. Objects could be allocated, used, and collected all within the cache, without ever being assigned a physical address. This is achieved with a software-controlled object-oriented cache. A data cache miss invokes a software interrupt that moves data between main memory and the cache.

To collect the young generation, roots are acquired from machine registers. A young set is used to track references from outside the young generation (the cache). This is preferred over the more typical old set, because the young set does not require accesses outside of the cache. The young set is maintained with non-local bits, which form a per-object mark set. Whenever a reference to an object left the cache, the software cache handler would set the non-local bit for the object. This performs the function of the generational write barrier, but only needs to be executed on cache line evictions.

The collector keeps the next generation in main memory, and collects it incrementally. The entire young generation is considered as live roots. This is reasonable since the cache is the young generation, and dead objects will tend to migrate out of the cache through normal replacement. Objects are either grayed or blackened as they are brought into the cache. Objects are blackened when the entire object is brought in at once; otherwise it is grayed. This is performed by the software cache handler on cache line fills. Marking is simply the process of cycling main memory through the cache. This amounts to using solution 1, but the read barrier need only run on cache misses. Because the barrier runs on all cache line fills, it is executed for *any* memory operation that misses the data cache, including stores and accesses to non-reference data.

Several concurrent GC algorithms use virtual-memory page protection to replace in-lined barriers. Appel, Ellis and Li [9] use this technique for a derivative of Baker's algorithm. The unscanned area is the portion of to-space containing unsafe objects whose references have not yet been redirected into to-space. Their algorithm read- and write-protects the unscanned area so that the virtual memory system performs the needed read-barrier check. Upon a page protection fault, the handler copies the offending page out of from-space into to-space, and updates all the references. This algorithm can cause a flurry of page faults at the beginning of GC, due to uncopied objects [21]. Such faults are also relatively expensive, since whole pages have to be copied and updated by the handler.

Boehm et al. use page protection to develop a concurrent conservative non-copying generational collector [21]. Copying is avoided because of the focus on conservative collectors. The algorithm is based on solution 3. This collector only write-protects pages, and performs only a small amount of work for each protection fault. Faults could be dispensed with altogether if hardware page dirty bits could be used. Before marking, all of memory is write-protected. For each write-protection fault, the collector notes the modified page and removes the write protection. After marking, objects missed due to concurrent modifications to the heap are detected by following all references from marked (live) objects on modified pages. This second mark can be done concurrently with the application if all the pages are re-protected to track further modifications. This can be repeated indefinitely, but eventually all modified pages must be marked while the application is stopped.

Boehm et al. go beyond proposing a particular algorithm, and propose a general transformation for making a wide variety of non-concurrent GC algorithms concurrent. The algorithm we developed can be considered a variation of this transformation. While they base this transformation on page protection bits, we use the RPA to provide the same basic functionality: the ability to track modifications to the heap. The RPA can collect a wider range of information at a finer granularity without causing interrupts. Hence, the RPA provides a basis for an even wider range of concurrent algorithms.

5.2. The Algorithm

As shown in Figure 3.3 of Section 3.2, the RPA replaces inlined write barriers by profiling store instructions. The RPA collects the needed information, and sends this information to write-barrier threads, which execute the write barrier in parallel with the application.

The resulting GC algorithm is a concurrent two-generation algorithm. Because moving objects concurrently is difficult [51], we focused on non-copying generational collection [21, 43]. The application must stop for three short pauses during garbage collection, as described in Subsection 5.2.4.

5.2.1. GC State

The GC algorithm uses three bits of each object's header word for GC state. The *LIVE* bit indicates the object has been reached during marking. Two other bits, the *OLDGEN* bit and the *REM* bit, are used for generational collection and are described in Section 5.3.

The GC algorithm uses load-locked/store-conditional primitives to atomically set GC state bits. However, the application does not require any fine-grain synchronization with the GC algorithm. Since the application does not modify or use the GC state bits, there are no synchronization issues between the application and GC.

The GC algorithm is based on the standard mark-sweep algorithm. Marking an object involves setting the *LIVE* bit, and placing the object on the *mark stack*. Objects are popped off the mark stack, and scanned for references to unreached objects, those that do not have the *LIVE* bit set. This process iterates until the mark stack empties, at which point all reachable objects have been marked. During sweeping, objects with the *LIVE* bit unset are collected and all *LIVE* bits are reset. Objects are allocated with the *LIVE* bit set when GC is marking, and unset otherwise.

```
store_bar:
mtp = load_locked(obj.mtp);
live_mtp = mtp | LIVE;
if (mtp ≠ live_mtp) {
     if (!store_cond(obj.mtp, live_mtp)) goto store_bar;
     // Add to live set ...
}
```

Figure 5.2. Write barrier for concurrent modifications.

5.2.2. Concurrent Modifications of the Heap

As stated in Subsection 5.1.2, we use the RPA to implement solution 2 for concurrent heap modifications. The RPA collects the stored reference for every reference store, and passes this information to a write-barrier thread that performs the barrier. If the *LIVE* bit of the referenced object is not set, then the *LIVE* bit is set and the object is placed in the *live set*. The query engine simplifies the concurrent write-barrier code by doing the required null reference check.

In our benchmarks, at least 98.5% of the time the *LIVE* bit is already set, and no more work needs to be done. The resulting write barrier, shown in pseudo-code in Figure 5.2, is three instructions in the common case. Although this is shown in C-like code, each statement represents one machine instruction.

5.2.3. Heap Organization

The heap is organized like the "big-bag-of-pages" used in the Boehm-Weiser conservative collector [22]. Heap space is allocated in chunks, and each chunk contains an integer number of same-size blocks. Chunks are nominally 1KB. However, unlike many collectors using this layout, all chunks need not be the same size, and they only need be double word aligned. If blocks of a given size do not fit evenly into a chunk, slightly smaller chunks are allocated for that size.

Supporting all block sizes is inefficient, as there are frequently not enough objects of a given size to fill up a chunk. This is especially true for larger sizes. All block sizes divisible by eight bytes are supported up to 128 bytes; block sizes divisible by 16 bytes are supported up to 256 bytes. An object will use the smallest size block into which it fits. Objects larger than 256 bytes are allocated separately. This combination appears to yield good performance.

Unallocated blocks are placed in per-size singly-linked lists stored in the free blocks themselves. Object allocation is inlined in the application code, and involves popping a free block off the list. Except for the first two words of memory, used for the free list, the GC algorithm initializes memory to zero concurrently, so allocation results in an initialized block.

If no free block is available then a new chunk must be allocated and initialized. This more complex operation is performed through a call to the run-time system. If this expands the heap beyond a dynamically adjusted threshold, GC is invoked. Typically only the young generation needs to be collected. If tenuring expands the old generation beyond a dynamically adjusted threshold, the entire heap is collected.

The algorithm eliminates all synchronization hazards between allocation, which pops objects off of free lists, and sweeping, which places collected objects onto free lists. Freed objects are collected and placed on the free lists en masse while the application is paused, making fine-grain synchronization unnecessary. A similar optimization is performed in the very concurrent garbage collector [71].

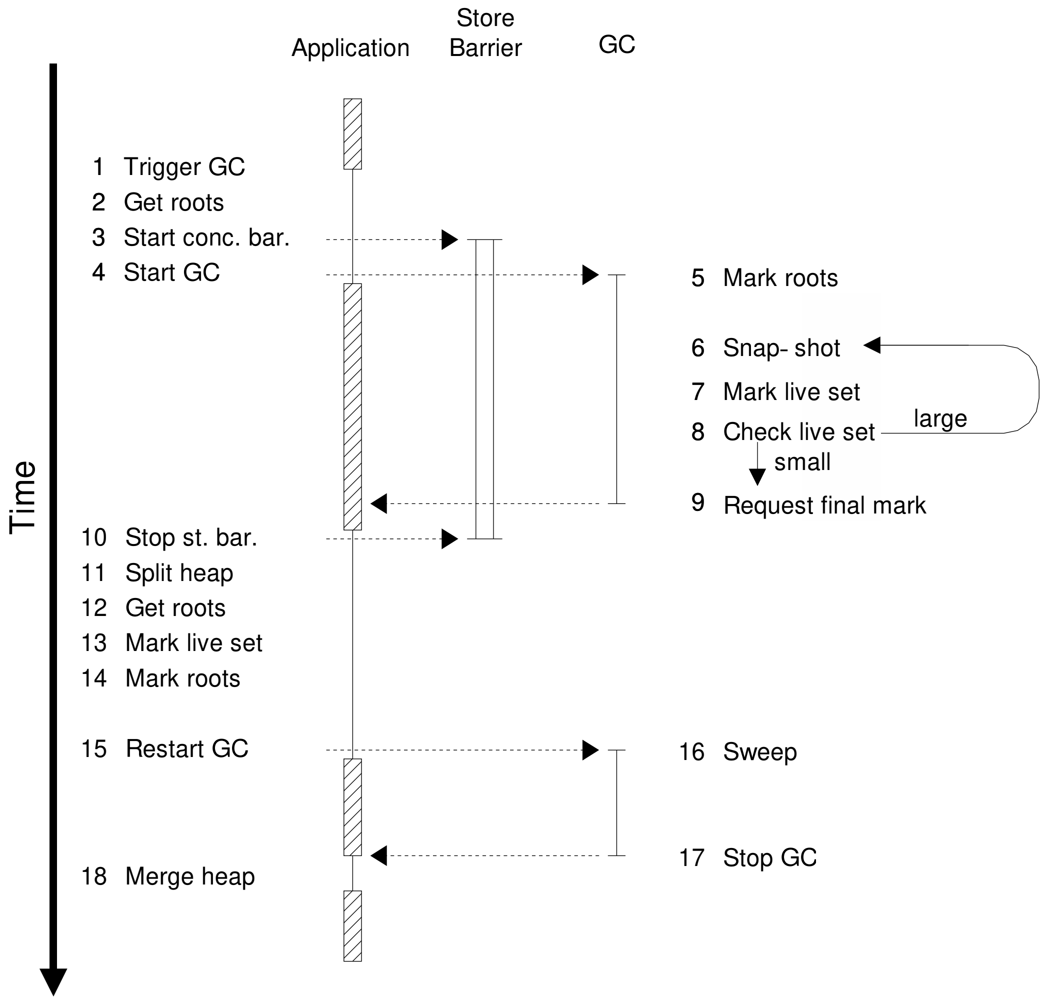


Figure 5.3. The concurrent GC algorithm.

5.2.4. Concurrent Algorithm

Figure 5.3 provides a time line of the algorithm's execution. Time runs down the image, and is not to scale. Four threads are shown, the application thread, two identical write barrier threads, and the garbage collection thread, which executes the marks and sweeps. The wide crosshatched bars indicate where the application is running. The narrow vertical lines indicate where the GC algorithm is running. The application is paused to perform GC-related tasks three times. No line indicates that the thread is not running at all. Horizontal dashed lines depict inter-thread control

The complete algorithm works as follows:

Steps 1-4

The first application pause occurs when the application tries to expand the heap beyond a threshold. This will begin the GC process. Next the application thread scans the call stack for root references. The service threads are ordered to begin executing the concurrent write barrier, required during marking, and the GC thread is ordered to begin the mark/sweep process.

At this point the application can be restarted.

Step 5

The GC thread marks the roots obtained in Step 2.

Steps 6-9

These steps take care of concurrent modifications. While the GC thread has been marking the roots in Step 5, the write barrier threads have been building the live set based on profiled stores, as described in Subsection 5.2.2. The GC thread atomically takes a snap-shot of the current live set to prevent the write barrier from modifying the live set that the GC thread is examining (Step 6), and then marks objects reachable from this set (Step 7). Write barrier threads find the live set using a global pointer within the run-time system. To snap-shot the live set, the GC thread makes a copy of the global pointer, and swaps in a pointer to another live set. Mutual exclusion locks make this operation atomic.

Like the page-protection based collector by Boehm et al. [21], the live set is marked repeatedly. While marking one live set the application and store barrier threads are still running, building a new live set. This new set can also be marked in parallel, but eventually the application must be stopped, and some amount of marking must be done with the program paused. The goal of a good algorithm is to minimize the duration of this pause. The size of the live set, built as explained above by the concurrent write barrier, is examined in Step 8. If the live set is large, it will be marked again in parallel. If it is small, a final mark with the application paused will be requested (Step 9). The threshold between "small" and "large" is doubled each iteration, guaranteeing the loop will terminate.

The size threshold starts at 20 objects, and is approximately doubled every iteration. The loop exits when the number of objects on the live set is less than the threshold. In addition, we find it very important to do at least one parallel mark, even if there are only a few entries in the initial live set. The number of entries only approximately indicates the amount of work to be done during marking, because a single reference could point to a large body of unmarked objects. This appears to happen quite regularly for the initial live set, so it is important to mark the initial live set in parallel regardless of its size. While the algorithm makes no guarantees, this does not seem to occur during subsequent passes through the loop. For subsequent iterations the number of live set entries is an accurate measure of the time needed to mark them. It is likely that the algorithm could be improved by aggressively reducing the live set to empty.

Steps 10-15

To perform the final serial mark, the application stops and pauses the write barriers in Step 10. This process ensures that there are no unexamined write-barrier messages in the system. In Step 11 the application thread (logically) creates a temporary object area to eliminate fine-grain synchronization between allocation and sweeping, as mentioned in Subsection 5.2.3.

The application thread then marks the final live set (Step 13), and obtains and marks the roots (Steps 12 and 14). At this point, all live objects have been marked. The GC thread is ordered to begin sweeping, and the application restarts (Step 15).

Steps 16-17

Sweeping collects dead, unmarked object (Step 16). The GC thread informs the application that it is finished, and stops (Step 17).

Step 18

The application, when preparing to allocate more memory, notices that GC has finished. At this point it (logically) merges the newly freed memory back into the heap for reuse. GC is complete.

5.2.5. Correctness

In this subsection we try to provide some intuition for the basis of the algorithm. GC correctness, defined as never collecting reachable objects, is difficult to prove in the presence of concurrent modifications by the application. Although this has been done formally for some algorithms [51], we must leave this for future work. The algorithm described here represents about 4000 lines of C code. Our feeling, echoed elsewhere [51], is that verifying the algorithm at a high, abstract level would not significantly improve our confidence in the implementation.

This algorithm assures correctness by monitoring moved references. The goal is to assure that a reference moved during marking will be found and marked no matter where it is moved. References may be moved from 1) local variables to the heap, 2) within the heap, or 3) from the heap to local variables.

Since roots, in local variables, are obtained with the application paused, these references will be captured at the beginning of GC. Hence, case 1 is not a problem. If the reference is moved within the heap, it will be placed in the live set by the write barrier threads, and marked in either Step 7 or 13. If the reference is moved to local or global variables, it will be caught when the roots are obtained the second time in Step 12, and marked in Step 14.

5.3. Generational Collection

The concurrent collector just described is also generational. Generational collectors [129] improve the efficiency of collection (in terms of work down per byte collected) by focusing collection efforts on recently allocated objects. Recently allocated objects, or *young* objects, tend to perish more quickly than older objects.

Generational collectors divide the heap into generations, where each generation contains successively older objects. Objects are first allocated from the young generation, which is collected frequently. Objects that survive a number of collections are *tenured* into the next older generation. Older generations are collected infrequently to find dead tenured objects.

This collector has two generations. Which generation an object is in is indicated by the *OLDGEN* bit in the object's header. When an object is tenured, the *OLDGEN* bit in the object's header is set. When marking only the young generation, the *OLDGEN* bit is examined to avoid marking old objects.

5.3.1. Generational Barriers

To mark only the young generation, all references into the young generation from the old generation must be found. This is done by using a *remembered set* to track all old-generation objects that may have references to young objects. The remembered set contains all old-generation objects that may contain references to young-generation objects. When marking the young generation, the old-generation objects in the remembered set are also examined, and references to young objects are marked.

A *generational write barrier* maintains the remembered set. When writing a reference to the heap, the generational write barrier examines the modified object. If the modified object is in the old generation then it is added to the remembered set.

The third and final object header bit, the *REM* bit, prevents the same object from being added to the remembered set more than once[129]. The *REM* bit is set when an object enters the remembered set. The write barrier examines the *REM* bit, and objects with their *REM* bit set will not be added to the remembered set again. To prevent young-generation objects from being placed in the remembered set all objects in the young generation also have their *REM* bit set, obviating the need for the generational write barrier to check the *OLDGEN* bit also. In the benchmarks we have observed, the *REM* bit is already set 99.99% of the time.

The resulting remembered set is free of duplicates, and contains exactly those objects in the old generation that must be examined. For most applications only a few old-generation objects point to young-generation objects, so only a few objects are placed in the remembered set.

5.3.2. Using the RPA

As with the concurrent write barrier, the RPA replaces inlined write-barrier code with RPA-based instrumentation. The RPA collects the needed information from selected instructions, and passes this information to service threads that perform the write-barrier function. The generational write barrier incurs no additional profiling overhead because the same instructions, stores of references to the heap, must be examined by both the concurrent and generational write barriers.

The RPA collects the base address of relevant stores— the base address being the object modified by the store. Write-barrier threads examine the *REM* bit of modified objects. If it is not set, the object is placed in the remembered set, and the *REM* bit is set.

The generational barrier code is similar to the concurrent store barrier code shown in Figure 5.2, and is also three instructions in the common case. Approximately two additional instructions are needed to remove the profile message from the message queue, resulting in a five-instruction generational write barrier.

The concurrent write barrier is extended to track both concurrent modifications and maintain the remembered set. It must be used whenever the collector is marking. The concurrent write barrier is twelve instructions in the common case.

5.3.3. Removing Objects from the Remembered Set

Objects are removed from the remembered set if, upon scanning them, no young objects are referenced. This creates a subtle synchronization hazard worth describing. A scenario is illustrated in Figure 5.4. The three steps represent an object in the remembered set with three fields being scanned.

In Step 1 of Figure 5.4 an object in the remembered set is being scanned from beginning to end for references to young-generation objects. In Step 2, the application writes a reference

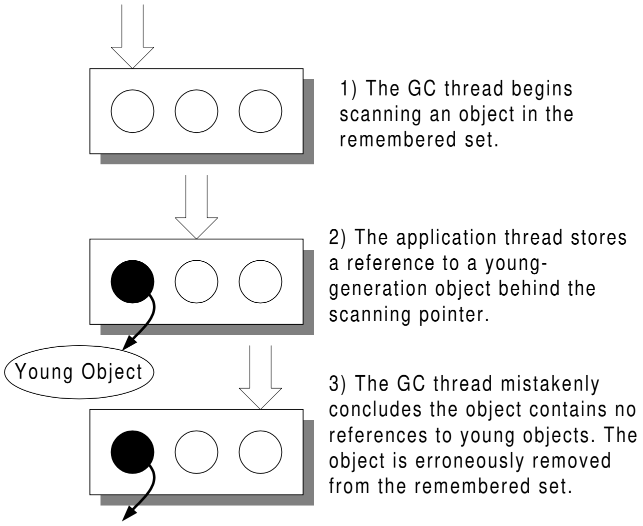


Figure 5.4. Removing objects from the remembered set creates a subtle synchronization hazard.

to a young object into the object. However, the GC thread has already moved past this reference field and does not see the new reference. In Step 3, the GC thread, believing the object contains no references to young-generation objects, erroneously removes the object from the remembered set. The referenced young object will not be collected during this GC, because the concurrent write barrier will catch it. It may be erroneously collected in the next GC, because the old-generation object is not in the remembered set and the collector will not find or follow the reference.

One solution is to remove the object from the remembered set before scanning it. In the scenario just described the generational write barrier would then add the object back into the remembered set.

If references to young objects are found, the object must be added back into the remembered set. When using a remembered set, it may be advantageous or necessary to protect against duplicate entries in the remembered set. In this case the GC algorithm must check to see if the object has already been placed back in the remembered set by the write barrier before adding it again. This check must be done atomically with placing the object in the remembered set.

5.3.4. Generational Heap Organization

Each generation is composed of a doubly-linked list of chunks; each chunk in the heap is either in the old or young generation. Whole chunks are tenured by unlinking them from the young generation and linking them into the old generation, in addition to setting the *OLDGEN* bits and adding each object to the remembered set.

Tenuring whole chunks wastes space since any free blocks cannot be used by new objects. The tenuring policy tries to minimize this waste by selecting nearly full chunks. Only chunks that are at least 75% full are tenured. In practice, between 0% and 14% of the tenured blocks are free. In addition, chunks are de-tenured when they become mostly free in order to reclaim otherwise wasted space. Blocks are de-tenured when no more than 25% full. Demers et al. describe a similar solution for their non-copying conservative generational collectors [43].

This process mixes old-generation objects into the young generation. Since old-generation objects must be explicitly skipped, sweeping the young generation becomes slightly less efficient. However, in practice less than 2% of objects in the young generation are old-generation objects.

5.3.5. Algorithmic Extensions

Generational collection adds four steps to the GC process. These steps are highlighted in Figure 5.5.

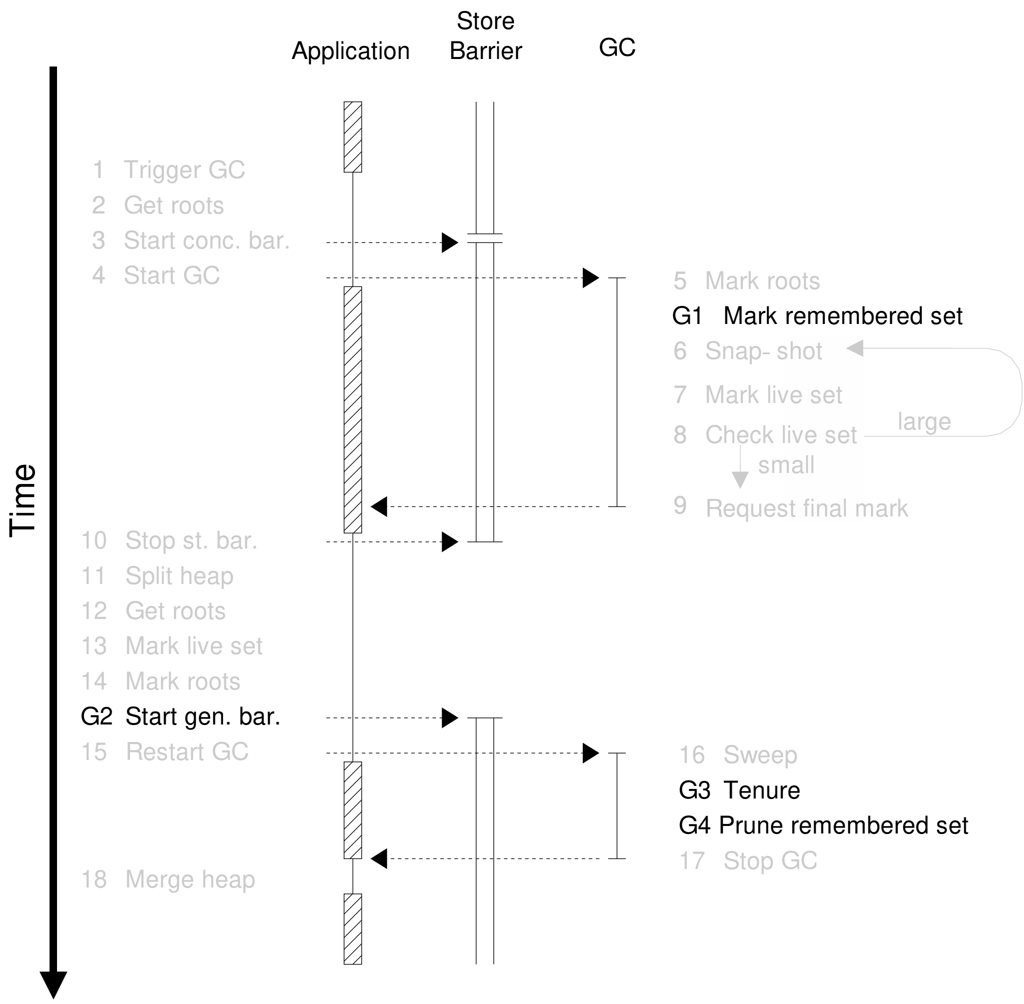


Figure 5.5. The generational GC algorithm.

Step G1

All objects in the remembered set are scanned for references to young-generation objects. These young-generation objects are marked. To prevent the write-barrier threads from modifying the remembered set while being examined by the GC thread, the GC thread atomically snap-shots the current state of the remembered set. During the rest of Step 6 the write-barrier threads will add objects to a spare remembered set. At the end of Step 6 the original and spare remembered sets are merged.

Objects remembered during this phase will not be examined during this step. This is not a problem because the concurrent write barrier will add any young objects so referenced to the live set, which is repeatedly marked in Step 7.

Step G2

Since the generational write barrier is needed whenever the application is running, the write-barrier threads must be restarted in generational mode in this step.

Steps G3-G4

Tenuring (Step G3) promotes (hopefully) long-lived objects into the old generation. When collecting the entire heap, objects in the remembered set may be freed, and these entries must be removed from the remembered set. This is done with an explicit scan of the remembered set following sweeping in Step G4.

5.4. Results

The results reported next are produced using the same methodology described in Section 4.4 to evaluate the RPA. Briefly, the application program is executed on an 8-way

Table 5.1. GC performance characteristics.

Bench	*GCs*	*Dispatch Stalls*	*Stall Overhead*	*Pause 1*	*Pause 2*	*Pause 3*	*Total*
		%	%	%	%	%	%
	(1)	*(2)*	*(3)*	*(4)*	*(5)*	*(6)*	*(7)*
compress	1	0.00	0.00	0.01	0.01	0.00	0.02
espresso	0	0.41	0.24	0	0	0	0.24
jack	5	4.60	3.52	0.06	0.07	0.07	3.72
java_cup	0	0.03	0.02	0	0	0	0.02
javac	2	0.15	0.00	0.02	0.03	0.02	0.07
jess	1	0.00	0.00	0.04	0.05	0.01	0.09
jlex	0	0.00	0.00	0	0	0	0.00
mpegaudio	0	0.00	0.00	0	0	0	0.00
pizza	0	0.75	0.18	0	0	0	0.18
raytrace	1	0.00	0.03	0.01	0.02	0.05	0.11
rpa	0	0.01	0.01	0	0	0	0.01
strata	3	0.10	0.02	0.04	0.06	0.11	0.23
xp	0	0.46	0.27	0	0	0	0.27
Average		0.50	0.33	0.03	0.04	0.04	0.38

out-of-order core. This is complemented by two service processors, which are six-stage in-order scalar pipelines. Each processor is capable of running three threads using fine-grained multithreading. The GC algorithm uses a single thread on one of the two processors. Write-barrier profile records are processed by two write-barrier service threads on the second service processor. Edge profiles are processed by a third service thread on the same processor.

5.4.1. GC Performance Overhead

Table 5.1 divides garbage collection overhead, the percentage of cycles the application spends on GC-related tasks, into two parts. Columns 2 and 3 describe performance losses due to the RPA. Columns 4 through 6 describe performance losses due to the GC algorithm. Seven of the benchmarks do not execute long enough to invoke a garbage collection (see Column 1). In this case, the algorithmic overheads are zero.

Column 7 combines these overheads, yielding a total GC overhead. Except for *jack*, the total overhead is less than 0.25%. The overhead for *jack*, although much higher, is still very impressive for a concurrent GC system. It is a mere 3.7%, caused mostly by RPA profiling stalls.

Columns 2 and 3 repeat the methodology of Subsection 4.5.2, showing first the percentage of cycles that dispatch was stalled due to the RPA (Column 2), and the resulting loss in performance (Column 3), when compared with an implementation of the RPA with much greater resources. As described in Subsection 4.5.2, there is some small error in these performance estimates for *db*, *espresso*, *javac*, *pizza* and *strata*. The out-of-order execution window covers about half of the dispatch stall cycles shown in the first column.

Columns 4 through 6 show the additional overhead caused by the short application pauses required by the GC algorithm. The first pause (Column 4) is for the initial collection of the roots, Steps 1-4. The second pause (Column 5) is for the final root collection and mark phase, Steps 11-17. The third pause (Column 6) is for returning collected memory to the application for reuse, Step 22. Collections are relatively infrequent and the pauses are short, leading to very low overhead. The *strata* benchmark has the highest algorithmic overhead of 0.21%.

Cache interference is another source of overhead not shown in Table 5.1. GC can interact with the application through the memory hierarchy, slowing the application down by causing extra cache misses. Should these effects be large, they can likely be reduced through algorithmic changes. For instance, service threads would not have to touch cache lines containing objects if the per-object GC state bits were kept separate from the objects themselves.

Directly comparing garbage collectors is difficult, due to variations in benchmarks, languages, and VMs. However, the time overhead for non-concurrent generational collectors is typically reported as 20-30% [10, 100, 145]. Concurrent collectors have much higher overheads when not run concurrently on another processor, often 100% or more [71, 83]. Concurrent collectors are used in this way to reduce pause times, rather than improve application performance. Domani, Kolodner and Petrank evaluate a concurrent generational collector on a multiprocessor system, but only compare the performance of concurrent collectors with and without generations [52]. O'Toole and Nettles compare a concurrent non-generational collector and a non-concurrent generational collector [100]. The concurrent collector slows down the application 12.8% on average, compared to the non-concurrent collector. Bacon et al. propose a novel concurrent collector based on reference counting and concurrent cycle detection [14]. This collector slows the application down by only 5% on average when compared to a parallel non-concurrent collector.

This collector also appears to be faster than explicit memory management using *malloc* and *free*, though direct comparisons are still difficult. Zorn et al. have made measured memory management costs for large allocation-intensive C programs [44, 146], and report overheads similar to non-concurrent GC, 20-30%. Note, however, that this is a comparison between *malloc*/*free* algorithms of a decade ago and a futuristic GC algorithm. It is likely that service threads and the RPA could also bring explicit storage management costs near to zero.

5.4.2. GC Phases

Table 5.2 shows execution times for various mark phases of GC, averaged over all GCs in each benchmark. The header for each column indicates which steps in Subsection 5.2.4 are being measured. Times are shown in thousands of cycles. For a 1GHz clock, this corresponds to microseconds. Only results for those seven benchmarks that performed a collection are shown.

Columns 2 and 6 show the durations of the first and second application pauses. These

Table 5.2. Mark phase execution characteristics. (thousands of cycles)

Bench	*GCs* (1)	*Get Roots Steps 1–4* (2)	*Mark Roots Step 5* (3)	*Mark Rem. Set Step 6* (4)	*Mark Live Set Steps 7–9* (5)	*Mark Roots Steps 10–17* (6)
compress	1	11	2149	1	5	13
db	2	9	1117	5131	11142	12
jack	5	15	542	5162	38	19
javac	2	16	1331	6945	312	21
jess	1	32	4819	1	1462	37
raytrace	1	12	2182	1	3338	14
strata	3	21	4470	4439	1119	28

Table 5.3. Sweep phase execution characteristics. (thousands of cycles)

Bench	*GCs* (1)	*Sweep* Step 18 (2)	*Tenure Step 19* (3)	*Prune Steps 20–21* (4)	*Merge Step 22* (5)
compress	1	1307	72	58	2
db	2	28111	998	795	31
jack	5	12373	598	559	18
javac	2	10275	467	360	14
jess	1	10227	510	436	10
raytrace	1	12541	493	445	39
strata	3	21805	2313	3123	55

pauses are uniformly short, having a maximum average duration of only 37 thousand cycles, 37μS with a 1GHz clock rate.

Columns 3,4 and 5 show the duration of the actual marking process. The remembered set is empty for the first collection, so benchmarks that invoke only a single collection spend little time on the remembered set. This produces the small values in Column 4. The initial mark of the root references takes between 0.5 million and 4.8 million cycles. The time to mark the live set varies dramatically, from 5 thousand to over 11 million cycles. The time taken to mark this set depends on what objects are placed on it. Long marking times occur when a large amount of live data is reached only through an object on the live set.

Table 5.3 shows the same information for the sweeping phase. Sweeping (Column 2), which zeros the memory for dead objects and links them into the free list, is clearly the longest phase. This is followed by a two relatively short phases. Tenuring (Column 3) transfers young objects into the old generation. Pruning (Column 4) removes unneeded entries from the remembered set.

Column 5 shows the duration of the final application pause time. This short pause adds the free lists of the newly reclaimed memory onto the free lists used by the applications. This tends to be the longest of the three pauses. The *strata* benchmark has longest average pause time of 55 thousand cycles or 55μS with a 1GHz clock rate.

Figure 5.6 represents similar information graphically for the first garbage collection in the *strata*. The total duration of the collection is a little less than 20 million cycles, shown on the X-axis. The upper time line shows the various GC phases executed concurrently by the GC and write-barrier threads. The lower line shows the application pauses. The pauses are so small that the hash marks plot on top of each other. As mentioned, the sweep phase is easily the longest phase.

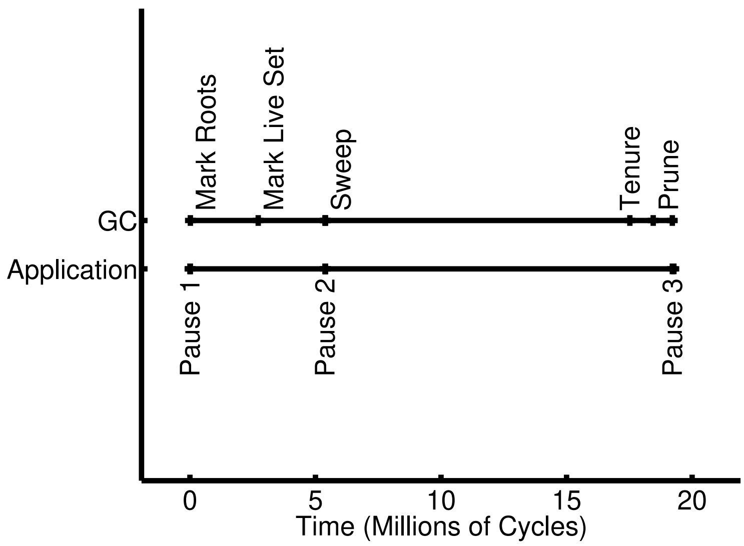


Figure 5.6. Time line for the first GC in the Strata benchmark.

5.4.3. Filtering Null-Reference Writes

Table 5.4 shows the effectiveness of the null-reference check performed by the RPA. Column 1 shows the percentage of profiled store instructions that stored a null reference. These stores are filtered out by the RPA, reducing the work to be done by the write-barrier service threads. These results do not include stores that write a null reference as an immediate value. The Strata compiler does not tag such stores to be examined by the RPA, since stores of null values do not need to be monitored.

Most benchmarks store few (computed) null references to the heap. The exception is *jack*. As discussed in Subsection 4.5.1, this is due to some inefficient software design in the benchmark. It is fortunate that the RPA can filter these null reference stores. Otherwise, the write-barrier threads would be completely overwhelmed by the Jack benchmark.

Columns 2 and 3 show the average number of cycles between each profiled store and each message sent. One message is sent for each profiled store, except for null stores. Note the clear effect on *jack*. Since most benchmarks rarely store null references, the numbers tend to be nearly identical. The system processes at most one message every 91 cycles (*db*).

5.5. Summary

Using the RPA and service threads, a concurrent generational GC algorithm was developed with processor overheads less than 0.25% typically and 3.7% in the worst case. Application pauses, a common problem with other GC algorithms, were never longer than 55,000 cycles or 55μS on a 1GHz processor.

This is accomplished by using the RPA to profile selected store instructions and pass this information to service threads through shared memory queues managed by the RPA. Using this information, the service threads perform a write barrier required by the garbage collector for correctness.

Table 5.4. Write-barrier work eliminated by the RPA.

Benchmark	*Cycles per Profiled Store (1)*	*Null Reference Stores (%) (2)*	*Cycles per Processed Store (3)*
compress	5.2	1.3M	1.4M
db	0.1	91.4	91.5
espresso	6.1	162.7	173.2
jack	95.0	14.4	286.8
java_cup	19.4	215.2	266.9
javac	9.4	243.1	268.2
jess	0.7	752.9	758.1
jlex	0.2	201.0	201.5
mpegaudio	0.0	768.3	768.3
pizza	31.3	127.8	186.1
raytrace	6.7	204.0	218.5
rpa	9.1	859.6	945.7
strata	15.0	183.3	215.7
xp	0.0	283.2	283.2
Average	14.2	111.2	257.3

Several features of the RPA made this achievement possible. Because the RPA can instrument instructions, guaranteeing that certain information is collected, the RPA can provide the information required for correctness by the GC algorithm.

The profile tag is used to select only those store instructions of interest to the garbage collector, those that store references to the heap. This is much superior to sampling all store instructions. First, it greatly reduces the number of instructions profiled. Second, it simplifies the task of the write barrier. If all stores were sampled, the write barrier would need to determine if the incoming data represented a reference stored to the heap, likely a complex task.

The RPA further reduces the workload for the write-barrier service threads by filtering stores of null references, which the GC algorithm does not need to examine. This reduces the bandwidth of the message stream, and simplifies the write-barrier code. This is crucial for one benchmark in particular, *jack*. Due to inefficient software design 95% of the references written by *jack* are null references. Were it not for the RPA's ability to filter these stores from the profile record stream, *jack* would completely overwhelm the write-barrier service threads.

The RPA, in cooperation with service threads, produces a write-barrier implementation combining the flexibility of software and the performance of dedicated hardware. This result demonstrates that the RPA provides a powerful set of primitives that VM software can use to build sophisticated profiling systems. Speculative object co-location provides a further demonstration of this expressive power in the next chapter.

CHAPTER 6

SPECULATIVE OBJECT CO-LOCATION

Java's object-oriented model provides a uniform view helpful to programmers. However, it can lead to inefficiencies when accessing objects. In particular, when two objects form a one-to-one parent/child relationship, the parent must reference the child indirectly, unlike languages such as C where the child structure can be embedded directly within the parent. Allocating the child and parent separately reduces cache locality, and the extra level of indirection leads to pointer-chasing overheads.

Speculative object co-location allocates multiple objects together to improve cache locality and reduce pointer-chasing overheads. Figure 6.1 shows two objects, X and Y, linked by a reference through field f. Without co-location (Figure 6.1a), X and Y are allocated separately at arbitrary addresses. Given a reference to X, Y is found by loading field f.

Initial investigations indicated that many of these relationships remained fixed throughout the lifetime of the objects. Only one X references each Y, and the two objects remain permanently paired. Often relatively few pairs are created, but the reference between them is traversed many times

Speculative co-location capitalizes on this stability as shown in Figure 6.1b to improve spatial locality and reduce the pointer-chasing overhead. The *co-located object*, Y, is placed within X, the *container*, at a prearranged location, *the co-location area*. The address of Y can then be obtained by adding a fixed offset to the address of X. Co-located objects remain intact. They are complete, valid object that can be referenced directly without complication. Maintaining this invariant simplifies static analysis and code transformation.

This work explores speculative object co-location in the context of the RPA. The RPA provides the needed profile information to determine which objects should be co-located. This investigation yields three insights into the operation of the RPA.


```
Y = LOAD [X + field_offset]
...
```

a) Without co-location.

```
Y = X + co_loc_offset
Y' = LOAD [X + field_offset]
if ( Y ≠ Y' ) Y = Y'
...
```

b) With co-location.

Figure 6.1. Object co-location improves spatial locality and reduces pointer-chasing.

1. Adaptive sampling. To enable speculative object co-location the RPA must profile *putfields*. Since *putfields* may execute only rarely, the sampling rate for these instructions is adjusted, using the profile tag to indicate the desired rate. Frequently executed *putfields* are given a low sampling rate to reduce overhead. Rarely executed *putfields* are given a high sampling rate to increase accuracy.

2. Task composition. Both the GC write-barrier described in the previous chapter, and the profiling for speculative object co-location described here need information from the same instructions. Furthermore, different information needs to be collected at different rates for different service threads. The RPA is capable of composing diverse tasks, even tasks with conflicting requirements for the same instructions.

3. Dead-locking. The dead-lock scenario, described previously in Subsection 4.3.1, was discovered while investigating speculative object co-location.

Our primary motivation in exploring this optimization is to examine novel uses for the RPA. Hence, we focus on profiling overhead in this preliminary investigation of speculative object co-location. Profiling is done on-line, and the overhead of this task is measured. Profile analysis and program transformation are done off-line, and the benchmarks are re-run using the same inputs. This obviously leads to optimistic results that place an upper bound on the expected improvement.

To separate issues related to speculative object co-location from issues related to the RPA, this research first develops an ideal version of speculative object co-location based on an ideal profiler. These initial results provide insight into the effects and operation of speculative co-location. Next, a profiler based on the RPA is constructed that produces essentially the same results, but runs with little overhead in most cases.

The next section provides an overview of speculative object co-location, dividing the process into four parts, profiling, reaching-definitions analysis, field selection and transformation. Only the profiling phase changes between the ideal version and the RPA-based version of speculative co-location. The ideal version is covered first. Section 6.2 describes the co-location transformation. Section 6.3 describes the ideal profiler. Section 6.4 describes the reaching definitions analysis. Section 6.5 describes the field selection algorithm. Section 6.6 and Section 6.7 describe the simulation methodology and results for the ideal version of co-location.

Section 6.8 describes the RPA-based profiler that replaces the ideal profiler in the RPA-based version of speculative object co-location. Section 6.9 and Section 6.10 describe the simulation methodology, overhead and effectiveness of the RPA-based profiling system. Section 6.7 discusses related work, and Section 6.12 summarizes this chapter.

6.1. Overview

Speculative object co-location co-locates paired objects, as shown in Figure 6.1. However, speculative object co-location does not *guarantee* that such relationships are stable. For

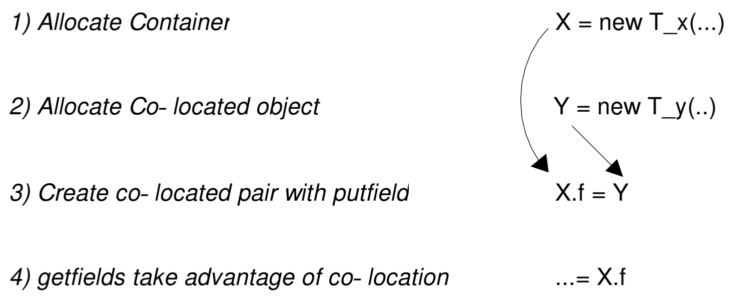


Figure 6.2. Speculative object co-location involves four points in the application code.

instance, the field f of Figure 6.1 may be modified to refer to an object other than Y. Therefore, the code dynamically checks the correctness of the speculative co-location offset, as shown in Figure 6.1b. Hence, only performance, not correctness, is effected should the co-located relationship change. In an out-of-order execution core, the check executes in parallel with the add and is off the critical path. The instruction overhead of this transformation can be reduced substantially, as discussed in Subsection 6.2.3. This technique essentially trades involved static analysis [49] for dynamic checking.

This implementation of speculative object co-location decides what to co-locate on a per-*field-declaration* basis. Each Java class declares the fields that constitute each object of that class. If a field declaration is selected for co-location, that field will be co-located for every allocated object of that class.

This includes objects of *subclasses*. A subclass extends another class, the *superclass*. Subclasses contain all the fields in their superclass, plus possibly additional fields. If a field in the superclass is co-located, then the same field will be co-located in all subclasses.

Objects are co-located as they are allocated by transforming the allocation sequence. Objects are not moved dynamically, though this may produce additional benefit.

Speculative object co-location involves the four points in the application code shown in Figure 6.2. The *container allocation site* creates the container. The *co-located object*

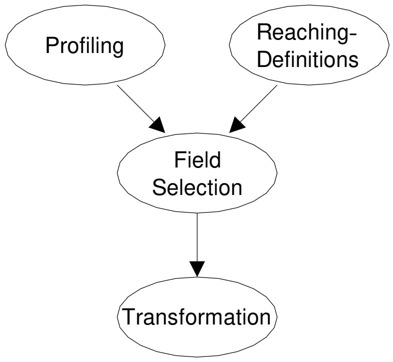


Figure 6.3. The speculative object co-location transformation is driven by three analysis steps.

allocation site creates the co-located object. The *putfield* creates a co-located pair by storing a reference to the co-located object into the container. Later *getfields* take advantage of the co-location.

Speculative object co-location determines which fields should be co-located, finds the associated points in the code, and transforms them appropriately. The entire process is summarized in four steps in Figure 6.3. Profiling determines which fields are beneficial to co-locate. For each field, the *getfields*, *putfields* and the container allocation sites can be trivially found since Java's type-safe bytecodes explicitly encode this information. Finding the allocation sites of the co-located objects is the purpose of the reaching-definitions analysis.

Field selection combines the results of these first two steps to produce a final list of fields to co-locate, along with the associated co-located object allocation sites, and the type and size information of potentially co-located objects. Field selection performs a series of tests on each field; fields that pass all the tests will be co-located.

Allocation sites are transformed so that co-location occurs as directed. Associated *getfields* operations may or may not also be transformed as shown in Figure 6.1. This measure-

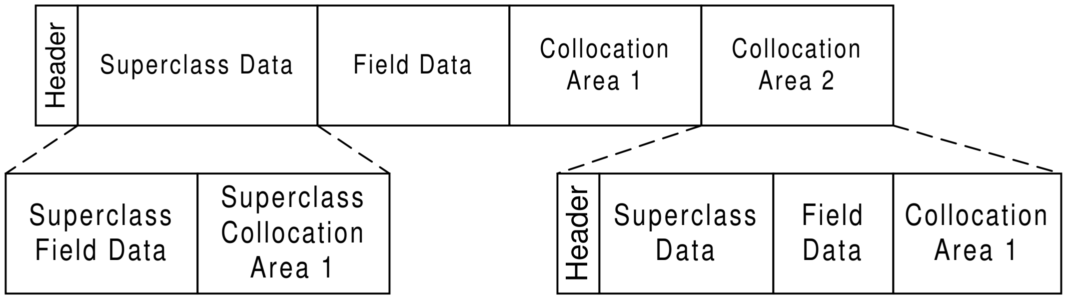


Figure 6.4. Co-Location areas are added to container objects to make room for future co-located objects.

ment technique separates the effects of enhanced ILP due to transformed *getfields*, from the effects of improved data locality due to co-located objects.

6.2. Speculative Object Co-Location Transformations

Speculative object co-location involves three transformations driven by information obtained from the field selection process. First, co-location areas are added to container objects to make room for co-located objects. Second, allocation sequences are modified to cause co-location to occur. Finally, *getfields* of co-located references are modified as shown in Figure 6.1 to take advantage of the co-location. As an experimental technique, the first two transformations can be done without the third.

6.2.1. Transforming Object Layout

To prepare for the co-location of objects, container objects are expanded by adding co-location areas. This is done as shown in Figure 6.4 by adjusting the allocated size of the container object, and adjusting field offsets as needed.

The initial word of each object is the header, used to store type information. This is followed by the data defined by the object's superclass. This data may include fields, as well as co-location areas for fields defined in the superclass. The fields and co-location areas for

```
1)   x = new T_x(...)               // Allocate container x of type T_x
2)   T_x.f.area = x + areaOffset;   // Set global pointer

...

     // Allocate co-located object y of type T_y
3)   if (T_x.f.area != null) {
          // Allocate from co-location area
4)        y = T_x.f.area;
5)        T_x.f.area = null; // Make sure the same area is not reused
          // Setup object header
     ...  }
else {
     // Allocate from heap as normal
6)        y = new T_y(...);
     }

...
     // Tie co-located object to the container
7)   x.f = y;
```

Figure 6.5. A global pointer is used to find available co-location areas.

the current class come next.

A single object may have multiple co-location areas, one for each co-located field. Note also that co-location can be nested. A co-located object may itself have co-location areas containing other co-located objects.

Field selection produces enough information to determine how big the co-location areas must be. For each co-located field declaration, the analysis phase provides the class of objects that may need to be co-located, and for arrays the maximum array size observed during profiling.

6.2.2. Transforming Allocation Code

When a co-locatable object is allocated, it needs to be co-located with the correct container object—the container whose co-located field will be set to refer to the co-located object. The correct container cannot be known exactly in general, since the *putfield* that ties the container to the co-located object has not yet occured. A simple heuristic selects a likely container candidate. Objects are co-located with the previously allocated container of the appropriate type.

This policy seems to work well, though obviously need not result in a correct pairing. In particular, this heuristic fails when the co-located object is allocated before the container. This scenario has been observed in our benchmark suite, and is discussed more in Subsection 6.3.2.

The policy is easily implemented using one global pointer for each co-located field declaration (Figure 6.5). When a container is allocated(x, Statement 1), the global pointers for co-located fields (*T_x.f.area*) are set to the appropriate co-location areas within the newly allocated container (Statement 2). When the co-located object is allocated, it examines the appropriate global pointer (Statement 3). If the pointer is not null, the object is placed in the co-location area, and the pointer is cleared (Statements 4 and 5). If the global pointer is null, then no container is currently available and the object is allocated from the heap as normal(Statement 6).

Since co-location can be nested, a co-located object may also be a container for other co-located objects. In this case the allocation site transformation is applied recursively, starting with the container transformation. Conceivably a single allocation site could become associated with multiple fields, resulting in a string of co-location area pointer checks. However, this case does not occur in the benchmarks we examined.

a) Original bytecode
```
      y = getfield x.f
```

b) Unoptimized assembly code
```
      ld y = [x+f_offset]
```

c) Internal representation with co-location speculation
```
      y = x+area_offset
      ld y' = [x+f_offset]
      y = SPEC y', y
```

d) Final assembly with co-location speculation
```
      y = x+area_offset
      ld y' = [x+f_offset]
      bne y, y', fixup
      cont:
      ...
      fixup:
      y = y'
      j cont
```

Figure 6.6. Pseudo-assembly representing the optimization of a *getfield*.

6.2.3. Transforming *getfields*

Since the co-location is speculative, the added offset is dynamically checked by performing the original load instruction, and then comparing the speculative value to the correct value. This is shown in Figure 6.6d. If the values do not match, a branch is taken to code that corrects the value.

This adds two instructions to the single original load instruction (Figure 6.6b). However, if the co-location transformation succeeds, later uses of y are only dependent on the add instruction, and may be dynamically scheduled earlier, resulting in more ILP. The loaded value is only used by the branch instruction. This branch should never be taken, and hence is easily predicted.

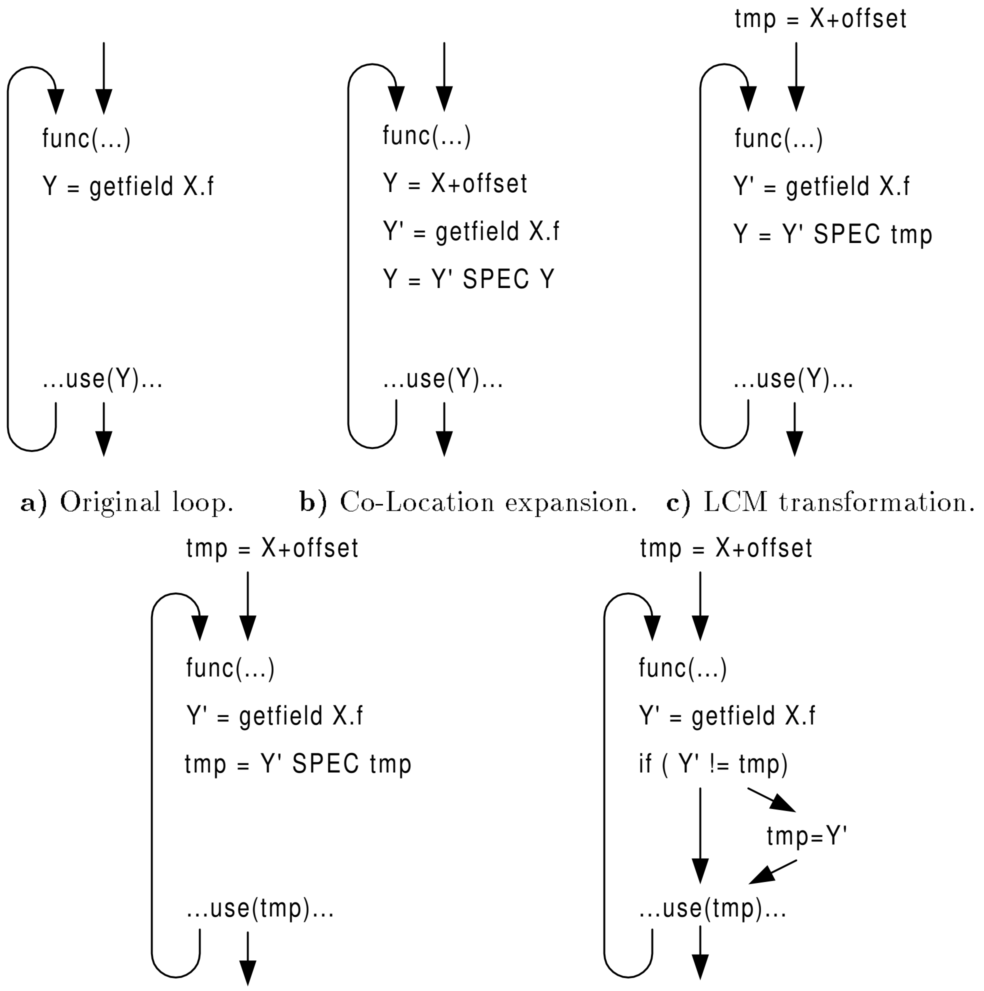
Strata's basic block layout algorithm ensures that the correct path is the not-taken path in most cases. In a few rare cases this layout algorithm uses the speculative check as the back edge of a loop, producing branches that are taken when the speculation succeeds. In the unoptimized version of the loop, an equivalent taken branch occurs elsewhere.

Figure 6.6c shows how Strata represents transformed *getfields* internally. Each *getfield* that accesses a co-located field is transformed into an add, followed by the original *getfield*, followed by a *speculative copy*. A speculative copy has two inputs, the *semantic source* before the *SPEC* keyword, and the *speculative source* following the *SPEC* keyword. A speculative copy has the semantics of a copy from the semantic source to the destination, but is implemented in assembly code using a conditional branch as shown in Figure 6.6d. This gives the code the desired ILP-enhancing characteristics. Essentially, the speculative source provides a prediction of the value of the semantic source. The speculative copy instruction may be useful for a wide range of compiler speculation, not just speculative co-location.

The transformation shown in Figure 6.6c is applied before lazy code motion (LCM), the primary global code motion transformation used by Strata described in Subsection 4.4.1. This is so LCM can optimize the sequence further, as shown in Figure 6.7. The branch is not expanded immediately because LCM does not move control-flow instructions. If the sequence were represented as a branch, it could not be hoisted.

The example loop of Figure 6.7a contains a function call that prevents a *getfield* from being hoisted out of the loop. However, after co-location is applied, the resulting add instruction can be hoisted, resulting in the code of Figure 6.7c.

In Figure 6.7c, the speculative source and destination variables of the speculative copy do not match, which is problematic. If the speculative copy is expanded in this form, an extra copy instruction will be required on the common path, to move the result of the add (*tmp*) into the result (Y). In many microarchitectures the copy is as expensive as the add,



a) Original loop. **b)** Co-Location expansion. **c)** LCM transformation.

d) Speculative destination renamed. **e)** Speculative copy expanded.

Figure 6.7. Transformation and optimization of a co-locatable *getfield*.

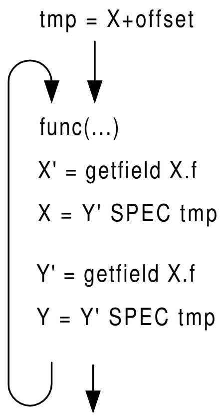


Figure 6.8. Optimizing speculative sources requires live-range analysis.

so nothing is gained by hoisting the add.

A special compiler pass transforms Figure 6.7c into Figure 6.7d, by renaming the destination of the speculative copy to be the same as the speculative source. In the example figure, Y is renamed to *tmp*. This transformation would be illegal were *tmp* used for anything other than a speculative source; *tmp* is a live variable that will be overwritten by the result of the speculative copy. However, since *tmp* is only used as a speculative source, the execution will be correct regardless of its value. Essentially, the transformation predicts that the speculative value propagated from the previous loop iteration is as good a prediction of the correct value as the co-location offset computed before entering the loop.

For this transformation to be legal, the live ranges of effected variables must not overlap. For instance, in Figure 6.8, X must not be live wherever Y is live, and neither must be live at the add instruction at the top of the loop. More aggressive optimizations are possible, for instance by using multiple speculative temporaries, but we have found the current algorithm effective. Furthermore, using only a single speculative temporary minimizes the number of additional live values that can hinder register allocation.

6.2.4. Required Invariants

Since the described form of co-location is speculative, little is required for correctness, beyond the check code. However, for performance the speculative checks should succeed in the common case. For this to be true, certain program operations must occur, and occur in the correct sequence.

The container object must be allocated first to ensure the co-location area pointers are correctly initialized. The co-located object must be allocated next, before another container object is allocated. A reference to the co-located object must be stored into the correct field of the container object. This field should not be changed and should be read often. Preferably, the accesses should be on the critical path of program execution. The criticality of the optimized *getfields* is a key issue for further study, and has not yet been addressed.

6.2.5. Integration with Garbage Collection

The interactions between object co-location and the garbage collector are fortunately simple. The only change is in reclaiming the space used by groups of co-located objects. A dead container cannot be reclaimed until all of the objects co-located inside the container are also dead. This is typically a simple change to the sweep phase of the collector.

This may lead to large amounts of wasted space if groups do not die all at once. This has not been seen in the benchmarks examined, but needs further investigation.

6.3. Ideal Profiler

An ideal profiler was developed to drive the preceding transformation. This defines an upper bound on the performance improvements of speculative co-location, provides a baseline against which the RPA can be measured, and facilitates exploration. The ideal profiler works by instrumenting the code with function calls at all important points. These are all *putfields* and *getfields* of references, and all object allocations.

6.3.1. Profiled Information

The ideal profiler emulates the co-location of objects, and then observes whether or not the co-location speculation at *getfields* would succeed. Simplistically, a field will be selected for object co-location if the co-location speculation is successful for a preponderance of the *getfields* that access the field.

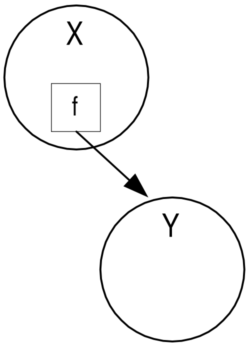
To emulate co-location, the profiler maintains some state for each object. Each reference field in each object may be in one of three states: *unassigned*, *co-located* or *not co-located*.

Fields of newly allocated objects are initially marked *unassigned*. Upon profiling the first *putfield*, the state of the field is changed to either *co-located* or *not co-located*, depending on whether or not the assigned object appears to be co-locatable (as discussed below). The algorithm implicitly assumes that only the first assignment to a field may be co-located. If another *putfield* is observed, the field is placed in the *not co-located* state. Throughout the profiling algorithm, silent *putfields* [84], which store the same value already present, are ignored.

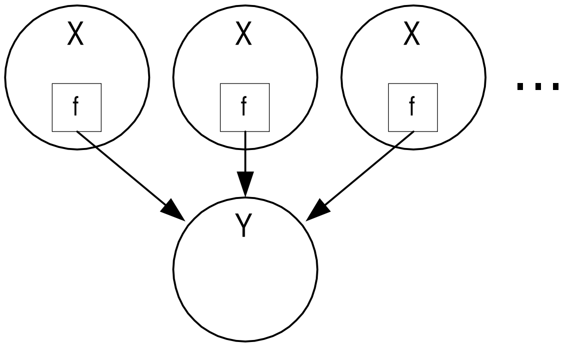
Upon profiling each *getfield*, the co-location state of the accessed field is examined. Fields marked *co-located* are counted as successes; all others are failures. Results are accumulated for each declared field, and fields with a high success rate and a high access rate are co-located.

6.3.2. Co-Location Topologies

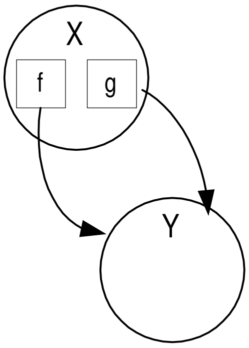
Some common data structure shapes inhibit or complicate object co-location, so the profiler identifies the forms shown in Figure 6.9. Figure 6.9a shows the desired simple case. Each container is co-located with a separate and unique co-located object.



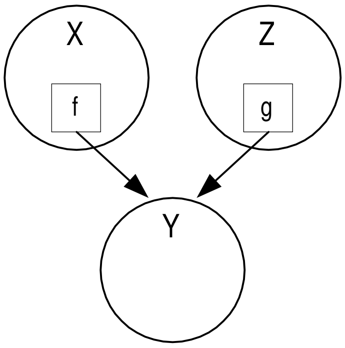
a) Simple readily co-located topology.



b) Same-field co-location topology. Multiple containers reference the same co-located object through the same field. Can not be co-located.



c) Same-object co-location topology. A single container references the co-located object multiple times through different field. Co-Located by creating a single co-location area for both fields.



d) Multi-object co-location topology. Multiple containers reference the same co-located object through different fields. Only one of the fields will be co-located.

Figure 6.9. Speculative object co-location uses profiling to distinguish different relationships or topologies between objects.

Figure 6.9b shows a common case that cannot be exploited by co-location. Many container objects reference the same co-location object through the same field. Since the co-located object can only be co-located with one of the containers, this topology is generally not exploitable. This arrangement often occurs when objects form a parent-child relationship in which the children have a reference back to their parent. In Figure 6.9b, Y would be the parent and the X's would be the children.

Figure 6.9c shows another surprisingly common case in which a container references a co-locatable object twice, through two different fields. This can occur due to subclassing. The subclass uses the co-located object for one purpose, and the superclass uses it for another. This has been noticed in the standard I/O libraries, for instance between subclass *java.io.PrintWriter* and superclass *java.io.Writer*. The same object is used by *java.io.PrintWriter* as the output stream and by *java.io.Writer* as a synchronization lock.

This topology can be co-located, provided the shape is stable and can be recognized. Only one co-location area is created, and both fields reference the one area. The co-location system presented here handles this case.

Figure 6.9d shows two containers referencing the same co-located object through different fields. This is distinct from Figure 6.9b, in which all the fields are the same. This can be co-located by combining all three (or more) objects together. However, this research does not try to exploit this case. Instead, profile analysis will co-locate only one of the two fields, the obvious choice being the one most frequently accessed.

6.3.3. Recognizing Co-location Topologies

The profiler uses the decision diagram in Figure 6.10 to distinguish between different co-location topologies. This is done to check for topology b upon profiling an otherwise co-locatable *putfield*. It is also done upon profiling co-locatable *getfields* , as described below.

To identify different topologies, the profiler records (hypothetically) co-located pairs in a mapping. The mapping contains the address of containers, the address of co-located objects and the field linking them. It can be accessed either by the address of the container or the address of the co-located object. This allows the profiler to detect when an object has been optimistically co-located to multiple places.

First, the mapping is searched for the co-located object. If the object is not found, then it has not been co-located elsewhere indicating an occurrence of the simple topology of Figure 6.9a. If the co-located object is found, then its container in the mapping is examined. If this container is the same object as the new potential container, then the topology of Figure 6.9c is detected. If the container object is not the same as the new potential container, then the field is examined. If the field in the mapping is the same as the new field, Figure 6.9b is detected, which cannot be co-located. If the fields are different, Figure 6.9d is detected, which possibly can be co-located.

Since topologies like Figure 6.9b cannot be co-located, the object's field will be marked *not co-located* when this topology is detected. In every other case, the field will be marked *co-located*. Upon observing a successfully co-located *getfield*, occurrences of the topologies of Figure 6.9b and Figure 6.9d are also recorded. The profiler keeps track of both fields involved and the number of *getfields* involving those two fields.

6.4. Reaching Definitions Analysis

Recall from Subsection 6.2.2 that both the container and the co-located object allocation sites are transformed in order to cause co-location at runtime. Given a co-located field, the container allocation sites are trivially known from Java's type-safe bytecodes. A static analysis based on reaching definitions finds the allocation sites for the co-located objects.

The static analysis also finds the particular container allocation site associated with each

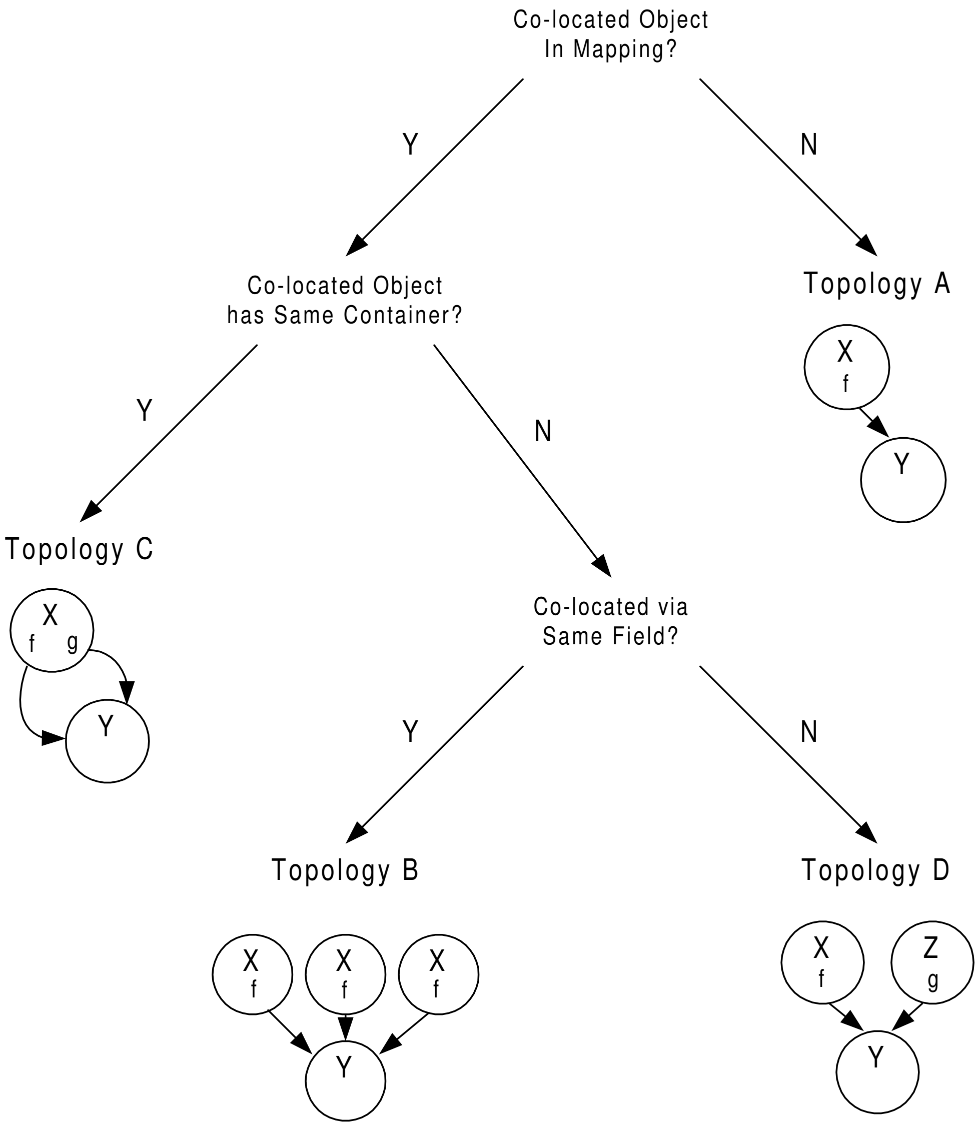


Figure 6.10. Decision tree used to recognize co-location topologies.

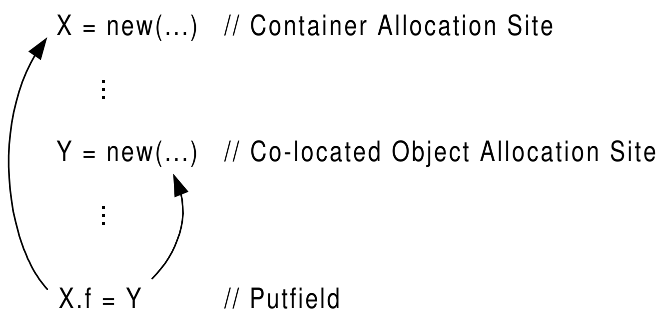


Figure 6.11. Reaching-definitions analysis locates allocation sites from *putfields*.

putfield. This information is used to determine the allocation order of the container and co-located object, as discussed in Subsection 6.5.8.

The reaching-definitions analysis is diagramed abstractly in Figure 6.11. The analysis starts with the *putfields*, easily found for a co-located field. When the *putfield* creates a co-locatable pair, the object modified by the *putfield* is the container; the value written by the *putfield* is the co-located object. Reaching-definitions analysis backtracks from the *putfield* to the associated allocation sites.

The analysis is interprocedural, but simple. The back-tracking need only cross two levels of method calls to find most cases that can be handled. Only reference values through local variables and call arguments are traced. The analysis gives up when it encounters more complex data-flow, such as through the heap. Hence, no alias analysis is needed. Heuristics based on profile information determine when the analysis is successful enough to warrant co-location.

6.5. Field Selection

Profiling, whether based on the RPA or ideal, produces data concerning the co-locatability of each field in the Java application. Field selection combines this with the static reaching-definitions analysis to produce the final list of fields to be co-located.

```
1)   Initial Filter
2)   Check Reaching-definitions Analysis
3)   Check Array Sizes
4)   Check for Cyclic Co-Location
5)   Topology Analysis
6)   Remove Insignificant Fields
7)   Check for Reversed Allocation Order
```

Figure 6.12. Field selection process.

Field selection follows the process shown in Figure 6.12 to determine which declared fields should be co-located. Each step in the process eliminates fields from consideration. Fields that survive the complete process are co-located. Field selection is identical for both the ideal profiler and the RPA-based profiler.

A simple performance model described in the next section guides the process. This model estimates the performance improvement to be gained from each field.

Step 1 eliminates fields that clearly cannot be co-located, because the co-location speculation was observed to fail frequently. These are typically fields that are modified frequently. Most of the fields in the application are eliminated in this step.

Step 2 checks the completeness of the reaching-definitions analysis. Since the reaching-definitions analysis cannot follow data-flow through the heap, it is sometimes incomplete, as described in the previous section. This step uses execution counts for *putfields* and allocation sites counts to estimate if the reaching-definitions analysis is complete enough to allow the field to be co-located.

Step 3 examines the profiled array sizes. Co-locating arrays that vary widely in size wastes space, because co-location areas are fixed at the largest observed size.This step eliminates fields where much space would be wasted.

Step 4 checks for various forms of cyclic co-location. Cycles occur, for instance, when an object should be co-located with another object of the same class.

Step 5 analyzes the frequency of topological conflicts observed by the profiler. Conflicts are resolved, often by co-locating only one of the conflicting fields.

Step 6 eliminates many unimportant fields. The step discards the least significant fields that, in total, account for less than 1% of estimated performance improvement.

Step 7 checks for reversed allocation order.

6.5.1. Performance Model

A simple performance model guides the selection algorithm. The relative performance improvement from co-locating a field declaration is calculated using the following equation:

$$improvement = successes - k_{fail} failures - k_{alloc} allocations \tag{6.1}$$

Equation 6.1 estimates that a successful speculation improves program execution time by one cycle. Each failure slows the program down by k_{fail} cycles. The penalty for allocation helps avoid certain pathological cases. Co-location increases the size of allocated objects. Strata, like many VMs, is not as efficient at allocating objects greater than a certain size. When co-location increases the size of an object beyond this size, 256 bytes, the extra overhead can swamp the benefits of co-location, if the field is not accessed often. For this research, k_{fail} is set at 5 and k_{alloc} at 20.

Equation 6.1 is used throughout the field selection process, which is now described in detail.

6.5.2. Initial Filter

The first step is to eliminate fields that are obviously not co-locatable. Equation 6.1 is evaluated for each declared field in the application using the number of successes, failures and allocations, as provided by the profiler. Fields which have an estimated improvement less than zero are removed from further consideration. Most fields are eliminated by this simple step, either because the reference field tended to vary, or because the field was involved in the topology of Figure 6.9b.

6.5.3. Check Reaching-Definitions Analysis

This phase merges the reaching-definitions analysis with the profile information. It determines if the reaching-definitions analysis was robust enough for the field to be co-located.

For both input operands of each *putfield*, the reaching-definitions analysis provides a list of the Java bytecodes that produce the value read by the *putfield*. In the desirable case, all the bytecodes are allocation sites. However, the reaching-definitions analysis may find a *getfield*, or it may not be able to find a source within two method call levels.

For a particular *putfield*, if the reaching-definitions analysis finds any source that is not allocation, then that *putfield* is considered poorly analyzed, even if some allocation sites are found as well. To determine whether a field is sufficiently well-analyzed, execution counts of all its *putfields* are totaled. Execution counts are provided by the profiler.

If the execution counts of poorly analyzed *putfields* total less than 20% of the execution counts of well-analyzed *putfields*, then the whole field is considered well enough analyzed for co-location. Fields not meeting this requirement are eliminated from further consideration. The threshold of 20% has not been studied extensively, but is not thought to be critical.

Typically, for co-located fields all *putfields* are completely analyzable—only allocation site sources are found. Often, those *putfields* that cannot be analyzed are never executed.

If a field passes this check, then the process produces a list of allocation sites that produce co-located objects for the field. This is the union of all allocation site sources for all *putfields* for the field.

6.5.4. Check Array Sizes

The next step examines array sizes. Since the system only creates fixed-sized co-location areas, co-located arrays must all be approximately the same size. The profiler provides the average co-located array size and the maximum co-located array size.[1] The co-location area is sized to fit the maximum array size, and the wasted space is estimated using the average size and the number of co-located arrays.

$$waste = (maximum - average) \times allocations \tag{6.2}$$

If the waste is greater than 30%, then the field is not co-located.

6.5.5. Check for Cyclic Co-Location

This phase examines the co-located fields for *cyclic co-locations*. These occur when, for instance, a field declaration would like to co-locate another object of the same class. Such cycles can be formed through subclasses, or through further co-location (e.g. an object of class X co-locating an object of class Y, which co-locates an object of class X). Only cycles based on subclassing have occurred in the benchmarks that were examined.

6.5.6. Topology Analysis

At this point, most fields are known to be co-locatable or not. However, fields involved in same-object or multi-object topologies need more analysis.

Given two declared fields, A and B, the relationship between these fields can be represented by ten values, five for each field. These are the number of allocations, the number of successful and unsuccessful co-locations and the number of same- and multi-object topolo-

[1] The ideal profiler profiles array sizes in a separate pass.

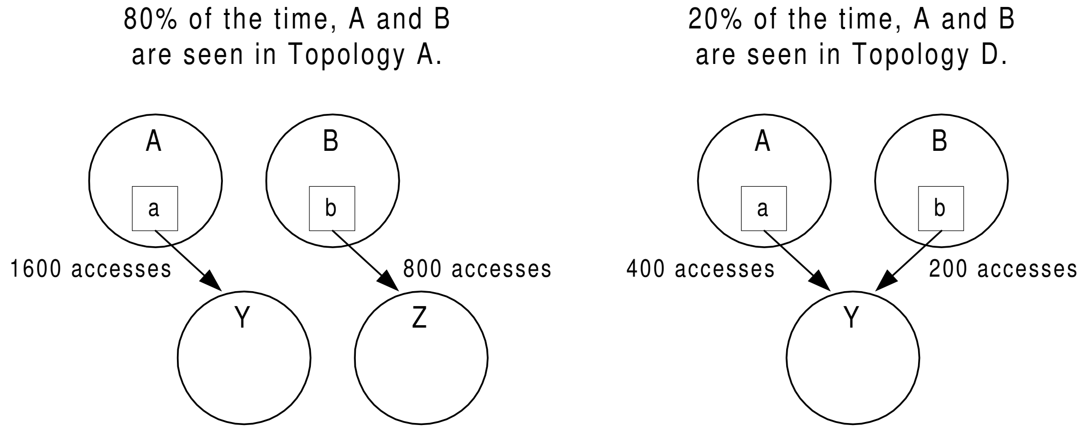


Figure 6.13. Example topological data.

gies accesses profiled from each field to the other. Topological information can be safely ignored if either field has already been eliminated from consideration. This greatly reduces the size of the problem.

As a concrete example of this analysis, consider Figure 6.13. In this example, two declared fields, A.a and B.b, are seen in a multi-object topology 20% of the time. A.a is accessed 2000 times in total, and B.b is accessed 1000 times in total. No speculation failures are observed.

There are several possible ways to co-locate fields A.a and B.b. Topological analysis computes estimated relative improvements for each case using Equation 6.1 and chooses the best one.

If neither A.a nor B.b is co-located the improvement will be zero. If either A.a or B.b alone is co-located, the topological data can be ignored, and the improvement can be calculated using Equation 6.1 directly. In this example, co-locating A.a only leads to an improvement of $2000 - k_{fail}0 - k_{alloc}5 = 1900$. Similarly co-locating B.b only leads to an improvement of 900.

If the multi-object topology occurs only rarely, it may make sense to co-locate both fields. Estimating the number of successes and failures for each field in this case is complex. There are two cases. In the event of a conflict only A.a or B.b will be able to co-locate Y. Which one gets the co-located object depends on which field is set first in the dynamic execution stream—which *putfield* executes first. If the *putfield* to A.a occurs first, then all accesses to A.a profiled as a success will be success. However, accesses to B.b profiled as successful, but involved in a multi-object topology with A.a, will be failures. This can be stated algebraically as:

$$\begin{aligned} improvement \quad = \quad & A_{succ} - k_{fail}A_{fail} + \\ & (B_{succ} - B_{soc} - B_{moc}) - k_{fail}(B_{fail} + B_{soc} + B_{moc}) - \\ & k_{alloc}(A_{alloc} + B_{alloc}) \end{aligned} \tag{6.3}$$

In this example, this will yield an improvement of 1800. If B.b is dynamically assigned first, the variables are reversed, giving an improvement of 1600. Since the order of *putfields* is not profiled, the average of these two cases is used, yielding a final improvement of 1700. This best case, co-locating A.a only in this example, is chosen by the analysis.

Similar reasoning can be applied for same-object topologies. This leads to five generalized improvement equations for selecting different cases. Details of these scenarios and their equations are omitted for brevity.

Unfortunately, a single field can conflict with multiple other fields, which makes the optimal decision dependent on the decision made for other conflicting pairs. This is solved heuristically using a greedy algorithm. Declared fields are sorted based on the number of profiled *getfields* in the presence of same- and multi-object topologies. Fields are processed one at a time in this order. For each field, its topological relationship with all other fields is considered, and a determination is made as to whether the chosen field should be co-located

```
class Main {
     parse(...)  {
          Scanner s = new Scanner(...);
          Parser p = new Parser(s);
     }
}

class Parser {
     private Scanner S;
     Parser(Scanner s) {
          S = s;
     }
}
```

Figure 6.14. Allocating the co-located object first is problematic for the current scheme.

or not. All following fields will be resolved based on this decision.

The heuristic algorithm does not necessarily choose a solution that maximizes the estimated improvement defined by Equation 6.1 for all fields globally. However, in the 14 benchmarks that we studied, few complicated cases occur, and the algorithm does in fact find an optimal solution in these cases.

6.5.7. Filter Insignificant Fields

This simple step removes many co-located fields that do not provide enough benefit to be worth optimizing.

6.5.8. Check for Reverse Allocation

When the co-located object is allocated before the container, this implementation of co-location will not co-locate the objects correctly, as mentioned in Subsection 6.2.2. This step determines the allocation order, and disables co-location on the field, if the co-located object is allocated first.

Figure 6.14 shows an example from *espresso*. Often, the co-located object (Scanner) is passed into the constructor of the container (Parser). This is usually done because the container does not know the exact type of object to allocate.

Even when the co-located object is allocated first, it still may be possible to perform speculative object co-location. Allocation of the co-located object is transformed to allocate space for a container, including a co-location area. Then the co-located object places itself within the co-location area of the object it just created. Container allocation is modified to check for the pre-allocated space.

This requires, in this implementation, that the exact type of the container be known when the co-located object is allocated. Unfortunately, this is not the case for the one important example of reversed allocation order seen in our benchmarks. Code from the *compress* benchmark is shown in Figure 6.15. Co-locatable objects i and o are allocated first, and then co-located with either a new compressor or decompressor, depending on which action is required. The container cannot be allocated early because the type of the container, either *Compressor* or *Decompressor*, cannot be known without further static analysis.

For each *putfield*, the allocation order of the container and co-located object is estimated with heuristics, based on the reaching-definitions analysis. When the co-located object is allocated first, co-location is disabled, since the one important case that occurred could not be exploited.

6.6. Simulation Methodology

The results reported in the next section are derived from the same infrastructure discussed in Section 4.4. However, the simulated configuration is changed to increase precision. The differences are as follows:

```
class Compress {
     spec_select_action(byte[], int action, int, byte[]) {
          i = new Input_Buffer(...)
          o = new Output_Buffer(...)

          if (action!=0) {
          x = new Decompressor(i,o);
          }
          else {
          x = new Compressor(i,o);
          }
     }
}
```

Figure 6.15. If the co-located object is allocated first, the container class must be known exactly.

1. The concurrent garbage collector is replaced with a non-concurrent collector, to reduce unpredictable interactions between the collector and the application.

2. No profiling is performed while evaluating the effects of co-location, to simplify interpretation of the results.

3. The measurement intervals are modified to ones more suitable for studying object co-location, rather than garbage collection.

The following subsections discuss these changes in detail.

6.6.1. Application Execution

Each benchmark application is compiled with three optimization levels, producing three distinct binaries for each benchmark. The performance changes resulting from co-location are measured by comparing the execution times of these three binaries. The binaries are produced by supplying different compiler options to the Strata compiler. The baseline binary contains no transformations related to co-location. Performance increase or decrease is measured against this binary. In the *memory-only* binary, allocation sequences are transformed for co-location, but *getfields* are unmodified. This differentiates the effects of changes in data layout, from the effects of *getfield* transformation. In the third binary, both allocation sequences and *getfields* are transformed.

The concurrent garbage collector is not used for these simulations, because it has caused random variations in performance, due to slight changes in the interleaving of memory accesses between the collector and the application. To eliminate this source of error, the concurrent collector is replaced by a non-concurrent mark-sweep collector. Garbage collections are sliced out of measured intervals, so that the collection process does not influence results.

6.6.2. Measurement Intervals

In the previous two chapters, a single large interval is measured near the beginning of long-running benchmarks. This is appropriate because the focus there is GC, and multiple small intervals would likely not capture complete GC cycles.

This methodology is not appropriate for studying co-location, because the effects of speculative object co-location are dependent on which fields are being accessed. Since different phases of program execution may exercise different fields, intervals are selected to assure that the measured portion of execution is representative. Intervals are selected automatically from functional execution traces describing which fields are accessed. These traces are produced by running an instrumented version of the benchmarks on a SPARC workstation.

The traces contain one record for each 100,000 heartbeats of application execution. Each record contains the number of accesses to each field during that period. For a set of measurement intervals, an estimate of the total number of accesses to a field during the entire execution can be made from the number of accesses during the measurement intervals. Comparing this estimate to the actual number of accesses during the entire execution, known from the trace, indicates how representative the intervals are.

For instance, if a set of three intervals contains 100 accesses to a field and represents 10% of the entire execution, we would estimate that the entire execution contains 1000 accesses. If the entire execution actually contains 2000 accesses, the intervals are not very representative.

The error for a set of intervals is computed by taking the absolute difference between the estimated and actual number of accesses, and summing over all fields. The set of intervals that minimize this error is chosen.

A simple algorithm tries all possible combinations of one, two and three intervals for each benchmark. The total number of heartbeats measured is held constant. When multiple intervals are taken, they are each of the same size.

Table 6.1 shows the number and size of each interval in heart beats. A short warm-up period that is added by hand to the beginning of each interval is also noted. Table 6.2 shows the size of these intervals in instructions, and Table 6.3 shows the size of the intervals in cycles. Reliable results for *pizza* are not available, due to register allocation problems described in Subsection 4.4.1.

6.7. Results

Speculative object co-location driven by ideal profiling reduces the execution time of six of 13 benchmarks by 0.04% and 8.0%, with an average of 2.7%. Two benchmarks run 0.6% and 1.8% slower. Five benchmarks have few *getfields* removed by speculative object co-location. This group runs from 0.7% slower to 1.7% faster, with an average improvement of 0.1%. As noted in the introduction to this chapter, these results are optimistic. The

Table 6.1. Measurement intervals (thousands of heart beats).

Benchmark	*Interval 1*			*Interval 2*			*Interval 3*		
	Fast Forw. (1)	*Warm Up (2)*	*Int*	*Fast Forw. (3)*	*Warm Up (4)*	*Int*	*Fast Forw. (5)*	*Warm Up (6)*	*Int*
compress	615	85	3300	19500	100	3300	36550	150	3300
db	674	26	1800	5080	20	1800			
espresso	97	3	Comp.						
jack	2184	16	1000	3284	16	1000	5283	17	1000
java_cup	81	19	Comp.						
javac	6383	17	2000	8870	30	2000			
jess	84	16	Comp.						
jlex	80	20	Comp.						
mpegaudio	1560	40	2700						
raytrace	75	25	Comp.						
rpa	5	5	Comp.						
strata	685	15	1100	18050	50	1100	25850	50	1100
xp	60	40	Comp.						

Table 6.2. Instructions in measured intervals (millions).

Benchmark	*Total (1)*	*Interval 1 (2)*	*Interval 2 (3)*	*Interval 3 (4)*
compress	177.0	53.5	67.8	55.6
db	287.5	159.0	128.5	
espresso	44.6	44.6		
jack	369.4	146.3	142.5	80.7
java_cup	29.7	29.7		
javac	279.0	151.6	127.3	
jess	252.3	252.3		
jlex	53.6	53.6		
mpegaudio	382.7	382.7		
raytrace	364.2	364.2		
rpa	8.0	8.0		
strata	451.7	129.2	158.1	164.4
xp	33.3	33.3		

Table 6.3. Cycles in measured intervals (millions).

Benchmark	*Total (1)*	*Interval 1 (2)*	*Interval 2 (3)*	*Interval 3 (4)*
compress	60.8	19.0	28.3	13.5
db	108.9	64.7	44.1	
espresso	16.7	16.7		
jack	122.1	45.5	49.3	27.2
java_cup	11.2	11.2		
javac	109.7	55.9	53.8	
jess	92.7	92.7		
jlex	16.2	16.2		
mpegaudio	82.2	82.2		
raytrace	128.7	128.7		
rpa	4.1	4.1		
strata	171.9	49.8	59.9	62.3
xp	11.4	11.4		

application is transformed off-line, and rerun with the same input set.

The remainder of this section discusses the types of fields co-located, and the effects of co-location on the baseline microarchitecture.

6.7.1. Observed Code Idioms

During the development of speculative object co-location, a number of code idioms were seen repeatedly. These are qualitatively described here.

Object Construction

Figure 6.16 shows the common simple case easily exploited by this implementation of speculative object co-location. The Face class of the *raytrace* benchmark represents one surface of a 3-D object. The container is constructed first. The co-located object is allocated and assigned inside the constructor of the containing object.

In almost all cases the *putfield* that ties the co-located object to the container occurs dur-

```
public class Face{
     private Point[] Verts;
     public Face() {
          Verts = new Point[4];
     }
}
```

Figure 6.16. Construction of a *Face* object in *raytrace* creates a co-locatable object.

ing the execution of the constructor. In most cases the *putfield* is directly in the constructor. In some cases the *putfield* is in a method called from the constructor.

Co-Located Arrays

Figure 6.16 also shows a common feature of co-located arrays. Often the array size is a fixed constant. In such a case the profiler will indicate the exact size, and there will be no wasted space.

Many other cases are similar to Figure 6.17 from the *jess* benchmark, in which the arrays size is passed in as an argument, available when the container is created. In such cases it is possible to allocate variable-sized co-location areas, like a tail-end array.

This would require static analysis to find the array's size within the code. It would also require checking, either statically or dynamically, that the container is not subclassed, since this would require the subclass's fields to be at a variable offset from the start of the object. Maintaining constant offsets for fields is typically important for simplicity and speed. Alternatively, extra space could be left before the co-location area for subclass fields.

While this seems to be a worthwhile improvement, this implementation of speculative co-location does not statically determine array sizes or allow for variable-sized co-location areas.

```
// Typical constructor code within Jess
v = new ValueVector(size);

public class ValueVector {
     protected Value[] v;
     public ValueVector(int size) {
          v = new Value[size];
     }
}
```

Figure 6.17. Array sizes are often available when the container object is allocated.

Enumerators

Figure 6.18 shows a common use of enumerators from the *strata* benchmark. Enumerators iterate over the elements held in a data-structure, such as the hash table in this case. Often, as in this case, only one enumerator for each hash table is active at a time. Since the enumerator has a reference to the hash table it iterates over, this forms a co-locatable pair. The enumerator is the container object, and the hash table is the co-located object. Essentially, the hash table has a single iterator prefixed to front of it.

This pattern is actually a special case of Topology B, in which multiple potential containers reference the same object through the same field. The difference here is that only one container exists at time.

This pattern, while common, is difficult to exploit. When allocating the iterator, the VM must verify that the previous iterator has died. This could be done normally, allowing the garbage collector to collect the object. However, this would allow only one co-location per GC cycle. Ideally, the VM should perform static analysis to prove that the previous iterator has died.

```
final public class Sparc extends Machine
{
     private Hashtable _stringImmeds;
     public void OutputImmediates(...)  {
          for (strImms = _stringImmeds.elements();
               strImms.hasMoreElements(); )
          {
               StringImmed lit = (StringImmed)strImms.nextElement();
          }
     }
}
```

Figure 6.18. Enumerators are common, but difficult to co-locate.

Co-Location in *java.lang.String*

The behavior of *java.lang.String* objects, used to represents character strings in Java, is a special case because these objects are so common. The exact implementation of the String class is not defined. In a typical implementation, the String object has a reference to a character array, as well as a starting location for the string within the array, and the length of the string.

This implementation has the advantage of allowing strings to share portions of a single array. For instance, if one string is constructed as a substring of another, the same character array is used, merely modifying the starting index and length. Unfortunately, this configuration leads to a lot of space overhead per string, typically 24 to 36 bytes, depending alignment requirements and the size of object headers.

The design also complicates co-location, since some values are shared, while most are not. Ideal profiling never selects *java.lang.String* for co-location. Profiling of *java.lang.String* is explicitly disabled in the realistic profiler.

It is likely that there is something to be gained from performing special optimizations for the handling strings within the VM. For instance, for small strings, the array can be combined with the object, the starting index can be removed, and the length field in the string can be combined with the length field in the character array. This would reduce the space overhead from 24 bytes in our VM to eight. Note that the overhead for a heap allocated string in C is five to nine bytes for *malloc*'s header and the terminating null character.

6.7.2. Co-Located Fields

Table 6.4 shows the number and type of fields co-located by the previously described process. Column 1 shows the total number of fields co-located for each benchmark, which varies between 1 (*db*) and 32 (*mpegaudio*). Generally, the performance improvements do not track the number of fields co-located. For instance, *db* benefits most from co-location even though only a single field is co-located.

This total is subdivided into application fields in Column 2, and standard libraries in Column 3. In most cases the bulk of the co-located fields are in application code. My experience with these benchmarks indicates that many of the fields that are co-located in the standard Java libraries are related to I/O. Many of these benchmarks are language tools and heavily use I/O for reading and writing files.

The total is also subdivided by the type of the co-located object. Column 4 shows the number of fields for which the co-located objects are not arrays. Columns 5 and 6 shows the number of fields for which the co-located objects are arrays. Column 5 shows those co-located arrays that are small, less than 100 bytes. Column 6 shows those arrays that are large, greater than 100 bytes.

While most fields target simple objects, co-location of arrays, especially large arrays, is quite common. The majority of the co-located fields co-locate arrays in five of the 14 benchmarks.

Table 6.4. Number and type of fields co-located, based on ideal profiling.

Benchmark	*Co-Located Fields*			*Co-Located Objects*		
	Total	*Application*	*Standard Lib.*	*Object*	*Small Array*	*Large Array*
	(1)	*(2)*	*(3)*	*(4)*	*(5)*	*(6)*
compress	8	8	0	3	1	4
db	1	1	0	1	0	0
espresso	9	6	3	1	1	7
jack	13	9	4	7	0	6
java_cup	16	0	16	5	6	5
javac	8	6	2	5	1	2
jess	20	17	3	16	0	4
jlex	9	6	3	4	2	3
mpegaudio	32	32	0	21	4	7
raytrace	9	8	1	5	3	1
rpa	17	11	6	10	3	4
strata	27	23	4	23	0	4
xp	4	4	0	2	0	2

6.7.3. Instruction Stream

Table 6.5 Provides basic statistics on the functional behavior of co-location. When multiple intervals are measured for a benchmark, the individual results for each measurement interval are shown, along with the totals for each benchmark. The benchmarks are divided into three groups. The first group contains six benchmarks that co-location helps. The second group contains two benchmarks that are degraded by co-location. The final group of five benchmarks show little opportunity for speculative object co-location, as measured by the percentage of dynamic instructions optimized by co-location (Column 2). While these benchmarks do improve very slightly, on average, these results tend to be in the noise. This subdivision of the results will be used throughout the remainder of this section.

Column 1 shows the percentage of optimized *getfields* for which co-location succeeded. That is, the percentage of times where the referenced object is located in the correct co-location area. For eight of the benchmarks, all such speculations succeed. For *jack*, the

Table 6.5. Co-Location instruction counts.

Benchmark	*Interval*	*Success Rate %* *(1)*	*% exec* *(2)*	*Instr. mem* *(3)*	*Added co-loc* *(4)*
compress	1	100.00	1.76	-0.05	1.73
compress	2	100.00	1.57	-0.15	1.86
compress	3	100.00	2.74	0.00	1.52
compress	tot	100.00	2.00	-0.06	1.68
db	1	100.00	0.74	-0.03	1.53
db	2	100.00	0.45	-0.62	2.93
db	tot	100.00	0.61	-0.22	1.99
jack	1	100.00	0.18	-0.48	2.18
jack	2	100.00	0.11	-2.43	2.50
jack	3	100.00	0.22	-0.28	2.33
jack	tot	100.00	0.16	-0.91	2.31
jlex	tot	98.24	0.22	-4.30	-5.54
mpegaudio	tot	100.00	0.38	0.02	5.32
raytrace	tot	100.00	1.46	-2.23	1.72
javac	1	99.77	0.47	-0.00	1.40
javac	2	95.73	0.05	0.61	4.63
javac	tot	99.44	0.28	0.05	1.67
xp	tot	100.00	0.36	0.04	4.00
espresso	tot	100.00	0.03	-1.53	1.89
java_cup	tot	70.10	0.04	1.15	4.70
jess	tot	100.00	0.03	-0.32	3.56
rpa	tot	100.00	0.07	-4.97	2.42
strata	1	100.00	0.04	0.22	2.29
strata	2	99.96	0.03	-1.43	2.75
strata	3	99.97	0.02	-0.61	2.56
strata	tot	99.98	0.03	-0.55	2.51

speculation failed an insignificant eight times. For all the remaining benchmarks, except *java_cup*, the success rate is quite high. *java_cup* has a field for which the container is sometimes allocated through a call to *java.lang.Object.clone()*. Cloning is another way of creating objects in Java, which speculative object co-location does not handle. The container is allocated via *clone()* 8% of the time, causing about 30% of the co-location speculations to fail.

Column 2 shows the percentage of dynamic instructions optimized. To compute this, the dynamic number of optimized *getfields* executed is divided by the instruction count for the base-case runs. The five benchmarks in the last grouping all show less than 0.1% of the instructions optimized by co-location. For improved benchmarks, co-location optimizes between 2.0% and 0.2% of the instructions. The two degraded benchmarks optimize less than 0.4% of the instructions.

Columns 3 and 4 show the expansion in the dynamic instruction stream due to co-location. This is measured as the change in instruction count with and without *getfields* optimized, divided by the number of optimized *getfields*. Hence, it is the additional instructions per optimized *getfield*. In those cases where few *getfields* are optimized, where Column 2 is small, this statistic becomes unreliable. Small changes in the instruction stream unrelated to co-location appear large relative to so few *getfields*.

Column 3 shows how optimizing the heap layout affects instruction count. Because multiple objects are allocated together, the allocation sequence is more efficient, resulting in reduced instruction count.

Column 4 shows the additional instructions, beyond the memory-only optimized runs, caused by the speculative co-location checks. Figure 6.19 describes these extra instructions in greater detail for the eight benchmarks significantly affected by speculative co-location. The figure divides the additional instructions into four categories, speculative co-location

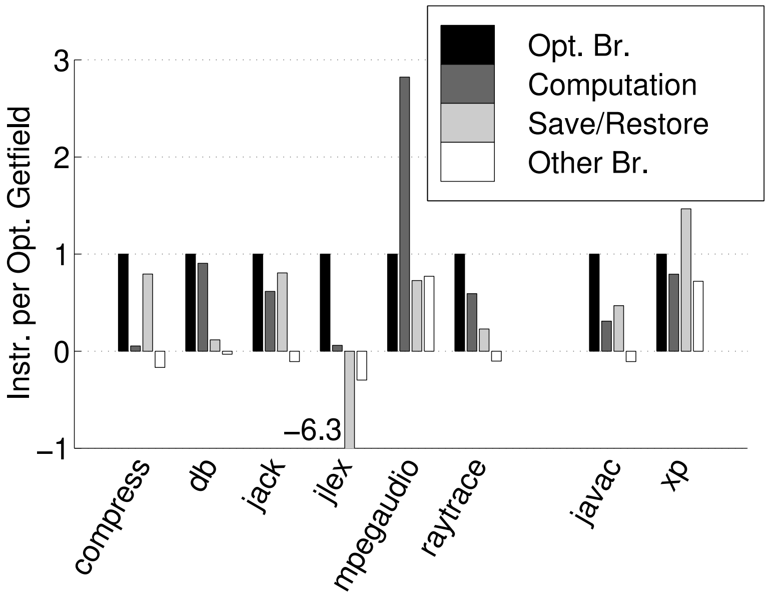


Figure 6.19. Additional instructions due to dynamic checking.

branches, additional computation, additional save/restore code and other branches.

Without further optimization, co-location adds two instructions to the stream for each optimized *getfield*, the add instruction and the branch. The first bar of each group shows the one additional branch that is always required. The add instruction may be hoisted out of loops (see Figure 6.7), reducing the additional computation instructions (second bar) below one. In *mpegaudio*the speculative checks are unfortunately disabling some of the analysis in Strata for array bounds check elimination. As a result, the optimized version of the benchmark executes more array bounds checks, increasing the computation overhead as shown. This limitation in the Strata compiler could likely be fixed.

The greatest overhead likely stems from increased save and restore code around function calls, shown in the third bar of each group. Typically, if the add instruction can be hoisted, but the load corresponding to the *getfield* cannot, then the load is being blocked by a function call. This scenario guarantees that the result of the speculative add instruction will be live across a function call. In a register-constrained situation, more variables will be allocated to caller-saved registers, resulting in more save and restore code around function call(s). These extra saves and restores significantly increase the cost of co-location; in fact they are likely the greatest cause of performance loss.

These saves and restores reduce performance in two ways. The most obvious cost is the increased instruction count. However, the saves and restores can also increase the data-flow path for computations. In our model a save and restore adds four cycles of latency to a computation. The additional saves and restores are not necessarily for the hoisted co-location temporary. Other computations may be adversely affected.

Such interactions between global code motion and the register allocator are not uncommon, and general techniques have been proposed to improve these cases [65]. While a number of improvements to our global register allocator have been made to reduce this effect, more improvements are possible. For instance, saves and restores can themselves be hoisted out of loops.

The *jlex* benchmark stands out with a *reduction* of over six instructions per optimized *getfield*. Examination of the benchmark's instruction trace indicates that this is an anomaly. The register allocator chose a less optimal register assignment for the base case, then it did for the optimized case.

The final instruction category is branches, shown in the fourth bar of each grouping. Adding the speculative checks modifies the control-flow graph within the compiler, affecting the basic-block layout algorithm. In most cases the change in the number of those branch instructions not related to speculative co-location is small. *mpegaudio* and *xp* both suffer from a slightly worse basic-block layout when fully optimized. Other benchmarks have a slightly reduced branch count.

6.7.4. Performance

Table 6.6 shows how speculative object co-location affects performance. Column 3 shows the total change in performance. Of the six improved benchmarks, five improved by between 0.3% and 8.0%, with an average of 3.2%. *jack* in this group does not improve significantly. The two benchmarks in the second group slow down by 0.6% (*javac*) and 1.8% (*xp*).

The change in performance is broken into two pieces. Column 1 shows the data-cache related effects. These are produced by only manipulating the heap without optimizing *getfields*. Column 2 shows the additional performance change from optimizing *getfields*. Column 3 just discussed is the sum of these two numbers.

We find the performance of co-location to be a complex interaction between the data memory access pattern, the register allocator and the ILP-improving effects of optimizing *getfields*. The final performance is the sum of these opposing effects. The following subsections discuss these effects.

6.7.5. Data-Cache Performance

Table 6.7 examines the relationship between execution time and the L1 data cache, as affected by object co-location. Column 1 shows the percent reduction in execution time, when co-location is performed without optimizing *getfields*. This data is duplicated from Table 6.6, Column 1.

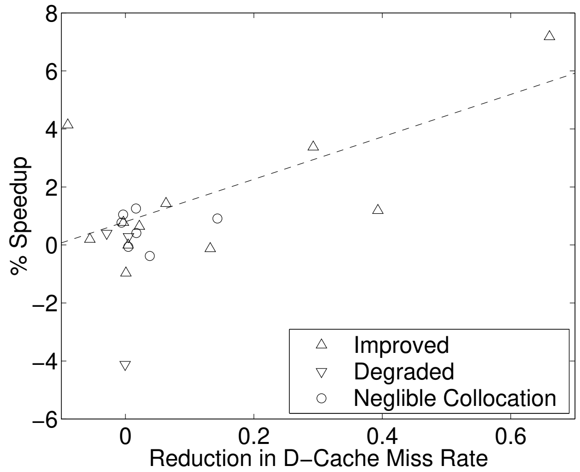
Columns 2 through 3 describe the data-cache behavior. The data-cache miss rate without co-location is shown in Column 2. Columns 3 and 4 show the change in data-cache miss rate resulting from object co-location without optimizing *getfields* (Column 2) and with *getfields* optimized (Column 3). Note that Columns 2 and 3 are nearly identical, as should be the case. Except for minor changes to call stack frames, optimizing *getfields* should not affect the data address stream.

Table 6.6. Performance.

Benchmark	*Interval*	*% Speedup*		
		Mem	*Col*	*Total*
		(1)	*(2)*	*(3)*
compress	1	-0.12	1.28	1.16
compress	2	-0.96	3.05	2.09
compress	3	1.19	-3.58	-2.39
compress	tot	-0.22	1.03	0.80
db	1	7.18	2.48	9.66
db	2	4.14	1.51	5.65
db	tot	5.95	2.09	8.04
jack	1	-0.00	-0.42	-0.42
jack	2	1.43	-1.13	0.31
jack	3	0.64	-0.39	0.26
jack	tot	0.74	-0.70	0.04
jlex	tot	0.78	1.91	2.69
mpegaudio	tot	0.20	0.09	0.29
raytrace	tot	3.38	0.90	4.28
javac	1	0.29	-1.26	-0.97
javac	2	0.40	-0.69	-0.29
javac	tot	0.34	-0.98	-0.64
xp	tot	-0.08	-1.68	-1.76
espresso	tot	0.41	-0.42	-0.01
java_cup	tot	-0.07	-0.19	-0.26
jess	tot	0.91	-0.40	0.51
rpa	tot	-0.38	-0.33	-0.71
strata	1	1.26	0.46	1.72
strata	2	1.05	-0.05	1.00
strata	3	0.77	0.13	0.90
strata	tot	1.02	0.18	1.20

Table 6.7. Data-cache effects.

Benchmark	*Interval*	*Mem*	*D-Cache Miss Rate (%)*		
		% Speedup	*Base*	*Base–Mem*	*Base–Col*
		(1)	*(2)*	*(3)*	*(4)*
compress	1	-0.12	5.29	0.13	0.12
compress	2	-0.96	8.06	0.00	-0.03
compress	3	1.19	1.12	0.39	0.39
compress	tot	-0.22	5.20	0.15	0.14
db	1	7.18	6.11	0.66	0.66
db	2	4.14	4.46	-0.09	-0.09
db	tot	5.95	5.37	0.32	0.32
jack	1	-0.00	0.49	0.00	0.01
jack	2	1.43	0.51	0.06	0.04
jack	3	0.64	0.40	0.02	0.02
jack	tot	0.74	0.47	0.03	0.03
jlex	tot	0.78	0.14	-0.00	0.00
mpegaudio	tot	0.20	0.31	-0.06	0.02
raytrace	tot	3.38	2.08	0.29	0.35
javac	1	0.29	0.76	0.00	0.01
javac	2	0.40	1.03	-0.03	-0.02
javac	tot	0.34	0.89	-0.01	-0.00
xp	tot	-4.12	0.72	-0.00	-0.01
espresso	tot	0.41	1.44	0.02	0.02
java_cup	tot	-0.07	1.25	0.00	-0.00
jess	tot	0.91	2.32	0.14	0.11
rpa	tot	-0.38	2.06	0.04	0.00
strata	1	1.26	0.63	0.02	0.03
strata	2	1.05	0.56	-0.00	0.00
strata	3	0.77	0.50	-0.01	0.00
strata	tot	1.02	0.56	0.00	0.01



Least-squares fit: Slope =7.31 Y Offset = 0.80 Correlation Coef. = 0.57

Figure 6.20. Correlation between speedup and data-cache miss rate.

Intuitively, the change in performance should correlate well with the change in data-cache miss rate. However, the data indicates this correlation is fairly weak for these small changes. For instance, co-location reduces the performance of the second measurement interval in *compress*by about 1%, but the cache miss rate is unchanged.

Figure 6.20 plots the speedup when objects are co-located, but *getfields* are not optimized, (Column 1) against the reduction in data-cache miss rate (negative of Column 3). A linear least-squares fit is also shown. Intervals from improved benchmarks are plotted with upward facing triangles, degraded intervals use downward facing triangles, and unaffected intervals use circles. The correlation coefficient of 0.57 indicates a fairly noisy correlation.

In an attempt to find a better measure of data-cache performance, total miss latency was also measured. Total miss latency is the total latency of all committed loads that missed the L1 data cache. However, changes in total latency correlates even more weakly with changes in performance.

To verify that the performance changes are in fact caused by changes in data-cache behavior, the L2 access time was reduced from 12 cycles to three cycles. This did, in

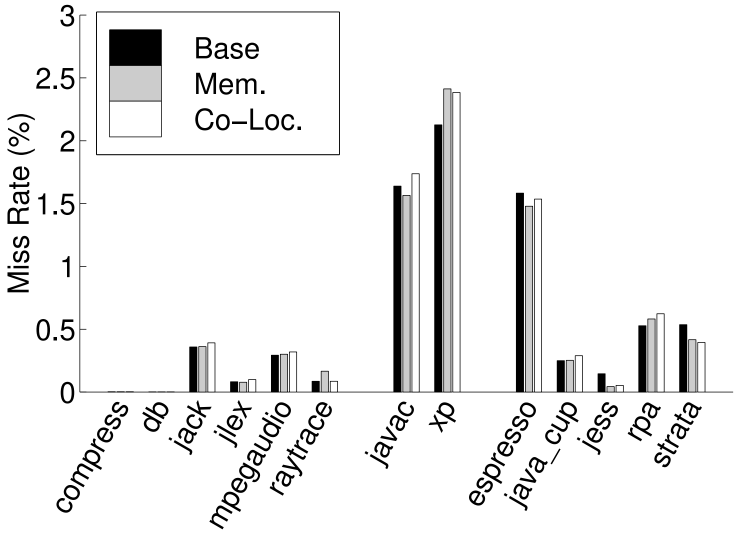


Figure 6.21. Instruction cache miss rates.

fact, eliminate the performance changes. However, performance is being influenced by more than mere cache miss rate. It seems likely that the remaining changes in performance are caused by a change in the relationship between cache misses. For instance, the fraction of missing loads that feed mispredicted branches may be changing, the data-cache misses may be moving between critical and non-critical instructions, or the overlap between cache misses may be changing.

6.7.6. Instruction-Cache Effects

Since co-location increases the static instruction footprint, it could conceivably increase the instruction cache miss rate. Fortunately, the instruction cache is not greatly affected by co-location. Those effects that do appear are caused by random shifts in the instruction address stream, not by code bloat.

Figure 6.21 shows the instruction cache miss rates for each benchmark, for the base optimization level, co-location without optimized *getfields*, and co-location with optimized

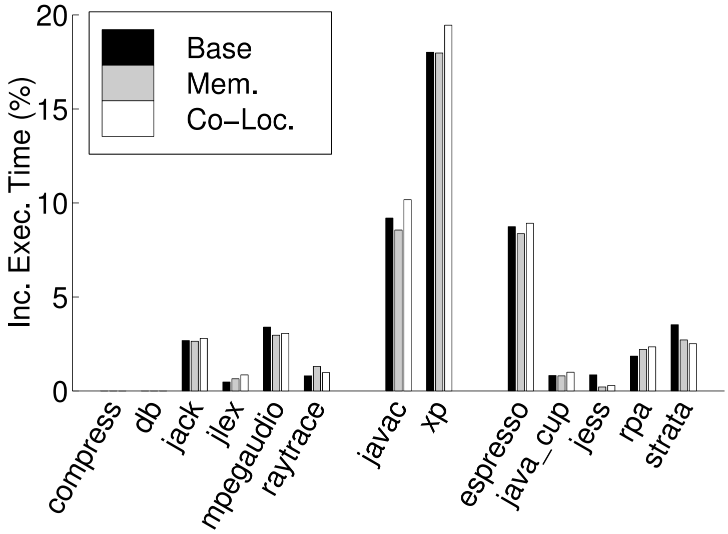


Figure 6.22. Performance lost due to instruction cache misses.

getfields. Figure 6.22 shows the performance loss due to I-cache misses for the same simulations. Performance loss is computed by comparing the execution time to a simulation in which the I-cache never misses. Performance is normalized to the ideal-I-cache case.

Most of the benchmarks, particularly the improved benchmarks, do not stress the I-cache greatly. Three of the benchmarks, *javac*, *xp* and *espresso*, are hindered by I-cache performance. The execution time *xp* increases almost 20% because of I-cache effects. Furthermore, this performance loss increases 1.5% when co-location is performed.

However, we find that this loss is not endemic to co-location. Rather, random variations in the instruction address stream cause minor changes in performance. Figure 6.23 shows I-cache miss rates, as the size of the cache is varied for the *xp* benchmark. The XP benchmark is right on the cusp of fitting in the I-cache. A 50% increase in cache size, from 32KB to 48KB, reduces the I-cache miss rate by a factor of six.

Figure 6.24 shows the effect on performance. At the base-case size of 32KB, around 20% of execution cycles are spent waiting on the I-cache. This performance loss falls off rapidly

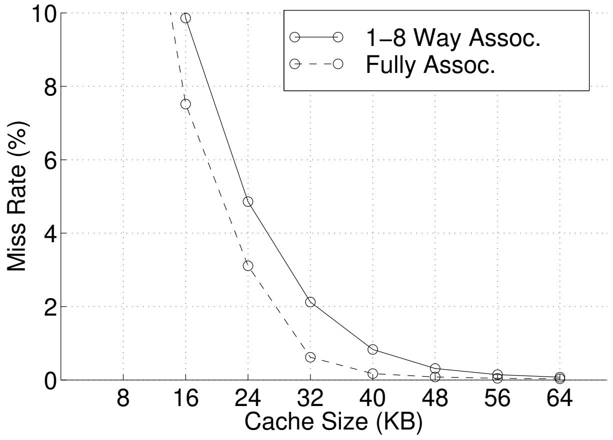


Figure 6.23. Instruction cache miss rates, vs. size for *xp*.

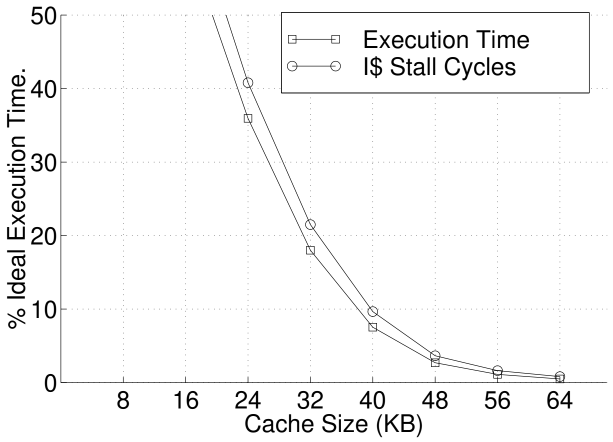


Figure 6.24. Performance loss due to I-cache misses for *xp*.

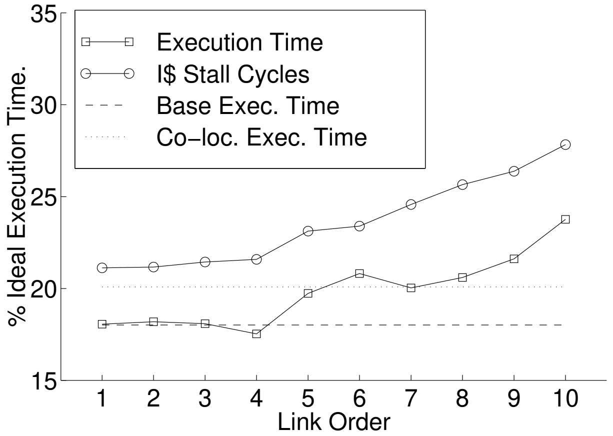


Figure 6.25. Performance variation due to object file link order for *xp*.

with increasing I-cache size, becoming negligible at 64KB.

The cycles that dispatch was stalled due to I-cache misses is also plotted. Dispatch stall cycles are normalized to the execution time with an ideal I-cache, so direct comparison to execution time is possible. The correspondence is clear. Some I-cache misses occur along misspeculated paths, so execution time does not increase quite as much as the increase in dispatch stalls.

This situation makes the performance of XP highly sensitive to minor changes in the instruction address stream. Figure 6.25 shows the performance of the base configuration, varying only the order that the object files are listed on the command line. This controls the order in which Java classes are laid out in memory. Only the object files of the application are reordered. The runtime system and standard libraries are not reordered. The graph is normalized to execution time with an I-cache that does not miss. The results are sorted by increasing I-cache dispatch stalls.

Merely varying the order of linking makes the execution time vary by more than 6%. The variation in execution time mostly tracks the change in I-cache-related dispatch stalls.

The small remaining variation is related to the prediction of function returns by the return address stack (RAS), which can be affected by the exact position of I-cache misses.

The two horizontal lines show the execution time of the base order and XP with object co-location enabled. The difference is readily explained by minor variations instruction address stream.

6.7.7. Branch Prediction

Although speculative object co-location increases the number of branches executed, it does not hurt branch prediction. Table 6.8 divides the branches into two groups. Columns 2 and 3 report on branches that are intrinsic to the application. Columns 4 and 5 report on branches that are added to optimize *getfields*.

Column 2 of Table 6.8 shows the base branch misprediction rate. Column 3 shows the *change* in branch misprediction rate with *getfields* optimized, although excluding those branches added for the optimization itself. Column 3 shows insignificant increases and decreases in branch prediction behavior for application branches. In many cases a small decrease in the branch misprediction rate is observed.

These results are sensible. Branches are only placed into the branch target buffer (BTB) when they are taken. Only branches in the BTB are explicitly predicted. All others are predicted not-taken by default. Since branches associated with object co-location are essentially never taken, they never enter the BTB, are never explicitly predicted, and use no prediction resources. Other branches are unaffected. The small changes observed maybe due to changes in the amount of aliasing, caused by shifts in branch PCs.

The conditional branches added by the co-location optimization are analyzed in Columns 4 and 5. Column 1 repeats the co-location success rates from Table 6.5, which gives a measure of the predictability of these branches. Column 4 shows the misprediction rate of

Table 6.8. Branch prediction effects.

Benchmark	*Interval*	*Success Rate*	*Branch Misprediction Rate*			
			Application Branches		*Co-Location Branches*	
			Base	*Base − Col*	*Succeeding Branches*	*Failing Branches*
		(1)	*(2)*	*(3)*	*(4)*	*(5)*
compress	1	100.00	4.17	-0.03	0	—
compress	2	100.00	5.34	0.01	0	—
compress	3	100.00	2.32	-0.11	0	—
compress	tot	100.00	4.04	-0.06	0	—
db	1	100.00	2.96	0.57	0	—
db	2	100.00	2.03	0.33	0	—
db	tot	100.00	2.57	0.47	0	—
jack	1	100.00	3.13	-0.06	0	100.00
jack	2	100.00	4.12	-0.05	0	—
jack	3	100.00	3.66	0.06	0	100.00
jack	tot	100.00	3.66	-0.03	0	100.00
jlex	tot	98.24	3.15	-0.02	0.02	3.69
mpegaudio	tot	100.00	1.50	0.04	0	—
raytrace	tot	100.00	3.67	-0.25	0	—
javac	1	99.77	3.40	0.03	0	2.24
javac	2	95.73	4.16	0.00	0	1.92
javac	tot	99.44	3.72	0.02	0	2.04
xp	tot	100.00	3.12	0.01	0	—
espresso	tot	100.00	3.44	0.03	0	—
java_cup	tot	70.10	3.47	0.02	2.27	0.63
jess	tot	100.00	3.92	-0.04	0	—
rpa	tot	100.00	5.27	-0.03	0	—
strata	1	100.00	4.45	0.02	0	—
strata	2	99.96	4.30	0.02	0.01	100.00
strata	3	99.97	4.28	0.00	0	100.00
strata	tot	99.98	4.34	0.01	0	100.00

these branches when the co-location speculation succeeds. When the speculation succeeds all of the time, all these branches are not-taken, and the implicit prediction rate is 100%. This is the case with most benchmarks.

Co-Location misspeculations lead to taken branches and mispredictions. However, results here are also encouraging. When a significant number of failures do occur, they tend to occur in bursts. When co-location fails for a heavily accessed object, a stream of misspeculations occur until the program moves on to another object. As a result, the branch predictor is able to accurately predict the failing cases too, which dramatically reduces the cost of speculation failures. The *javac* benchmark is a good example. The co-location branches are not-taken 99.5% of the time (Column 1). Yet the *taken* branches are predicted with 98% accuracy (Column 5). For *jack* and *strata* failures are so rare that the co-location branches are always mispredicted when the co-location speculation fails. *jack*, for instance, only had eight co-location failures.

6.8. Co-Location Profiling Using the RPA

Co-Location in the previous sections is driven by an ideal profiler, to gain a better understanding of the intrinsic benefits of speculative object co-location. The ideal profiler provides the best profile information possible, irrespective of overhead. As a result, applications run at least an order of magnitude slower under the ideal profiler.

The ideal profiler provides a standard against which to measure a more realistic profiler, based on the RPA. The RPA is used to collect information on *getfields*, *putfields*, and allocation sites. A single service thread consumes this information, along with the standard edge profile records. The rest of the co-location analysis, which includes the static analysis and the field selection algorithm, is independent of which profiler is used.

The theory of operation of the RPA-based profiler is similar to the operation of the ideal profiler. A number of improvements where made to yield a profiler more compatible with on-line profile analysis using service threads. The most obvious of these is replacing instrumentation with the RPA. In addition, an adaptive sampling rate algorithm is used to reduce the profiling overhead. Topology analysis is simplified, so that only *getfield* profiles are needed. The profiler still observes *putfields*, but only to estimate execution counts. Finally, the topology mapping data-structure is optimized for space and speed. Each of these optimizations is discussed in the remainder of this section.

6.8.1. The RPA Queries

The RPA assembly language queries used for the co-location profiler are shown in Figure 6.26. These queries include profiling for the concurrent GC algorithm, and the edge profiler.

There is overlap between the GC algorithm and the co-location profiler. Both need to profile the same store instructions, those that store references to the heap. However, the information is collected at different rates for different service threads. Hence the query sends the record to two places, the write-barrier service thread, and the co-location profile service thread.

The co-location profiler uses the profile tags for adaptive sampling, as explained in the next subsection. This leads to several sets of nearly identical queries.

Load instructions that correspond to *getfields* of references are tagged by the VM, just like stores that correspond to *putfields*. See Query 7 of Figure 6.26. Since accesses to co-located fields should be frequent, *getfields* are sampled at a medium rate of one sample in 64 instructions.

Profiling allocation sites is a little more complex, because there is no single instruction that naturally represents the allocation, unlike *getfields* and *putfields*. Instead, the Strata compiler inserts an NOP just for the purpose of having something to profile. Although

```
// Write-barrier profiling plus co-location putfields
1)   for opST 1 always collect op1 op2 op3 pc;
          send 1 goto writebar;
     writebar:
          if op1 <> 0 then send 0 stop else stop;

2)   for opST 2 always collect op1 op2 op3 pc;
          if filter 4 & filter 2 then send 1 goto writebar2;
     writebar2:
          if op1 <> 0 then send 0 stop else stop;

3)   for opST 3 always collect op1 op2 op3 pc;
          if filter 16 & filter 4 then send 1 goto writebar3;
     writebar3:
          if op1 <> 0 then send 0 stop else stop;

// Allocation site profiling
4)   for opALU 1 always collect op1 pc;
          send 1 stop;

5)   for opALU 2 every 8 collect op1 pc;
          send 1 stop;

6)   for opALU 3 every 64 collect op1 pc;
          send 1 stop;

// Getfield profiling
7)   for opLD 1 every 64 collect op1 op2 pc;
          send 1 stop;

// Edge profiling
8)   for opBRANCH * every 256 collect pc misc ;
          send 1 stop;
```

Figure 6.26. RPA queries used during object co-location profiling.

this instruction does not affect register state, it can still be profiled. The NOP takes as input the desired piece of information, the array size. The query of Figure 6.26 collects this information and the PC to identify the allocation site.

Messages for *putfields*, *getfields*, allocations and branches are all sent to the same message queue and consumed by the same service thread. The service thread must distinguish between the different types of records it receives. The RPA places the PC of the query that sends a message into the header word of the record, and the service thread uses this information to determine the content of each record.

6.8.2. Adaptive Sampling

Often co-located pairs are rarely created, but frequently accessed. This complicates the profiling of events associated with the creation of co-located pairs like *putfields* and allocation sites. For example, *compress* allocates only three instances of co-located objects, but accesses these objects many times. Profiling one *putfield* in 64 would likely sample no *putfields* associated with co-locatable objects, defeating the co-location analysis. Experiments confirm this behavior.

For this reason, the profiler uses an adaptive sampling rate. A high rate accurately profiles infrequent instructions; a low rate reduces the profiling overhead for frequent instructions. The profile tag encodes the desired sampling rate for each instruction.

Consider the allocation site queries, Queries 4–6 of Figure 6.26. Initially, the profile tags of all instructions are set to one. Query 4 will handle these instructions, performing no random sampling. After the profiler receives 16 records for the same instruction, it increments the profile tag to two. Query 5 handles opALU instructions with a profile tag of two. Random sampling selects only one in eight of these instructions. When the service thread estimates that an instruction has executed 256 times, the profile tag is incremented to three, reducing the sampling rate to one in 64.

Since the same instruction may be profiled at different rates, the profiler needs to know the sampling rate to estimate the execution count. Samples generated by Query 4 represent a single execution. Samples from Query 6 represent 64 executions. The profiler determines the sampling rate using the header word of the record. The header indicates the query that sent the record, and the query determines the sampling rate.

6.8.3. Topology Recognition

The ideal profiler observed both *getfields* and *putfields* to determine the topological relationships between objects. The RPA profiler only uses *getfields* for this purpose. Essentially, the *putfields* are inferred by observing changes in the values retrieved by the *getfields*. Otherwise, both the RPA-based profiler and the ideal profiler use the same topology recognition algorithm, the decision tree of Figure 6.10.

However, the ideal profiler maintains the mappings between containers and (hypothetically) co-located objects in a standard hash table. This data structure was optimized for the RPA-based profiler to reduce the amount of storage required and to accelerate accesses to the mappings.

The mappings are maintained by two software caches totaling 48KB. The *container cache* holds the addresses of container objects and the state of the fields in those objects. The *co-located object cache* holds the objects potentially co-located with these fields. Each cache is indexed by object addresses. Objects in one cache contain a link to the associated object in the other cache. This provides enough information to determine object topologies.

The cache has room for 2K containers, 8K container fields and 4K co-located objects. This is insufficient to cover the entire heap. The number of objects that are placed in the cache is limited by object class. Each class in the application can have at most ten objects in the cache. This makes sure most fields in the application are analyzed, without overwhelming the cache.

6.9. Simulation Methodology with RPA-Based Profiling

The results in the next section evaluate the quality of the RPA-based profiles, the overhead of collecting the profiles, and the change in performance when the profiles are used for object co-location. The performance of the co-location transformation is determined using the same methodology as the ideal results. The following two subsections describe how the RPA-based profiles are collected and how the profiling overhead is measured.

6.9.1. Collecting Profiles

To be comparable to the ideal profiling results, profiles must be collected over the entire execution of each benchmark. However, simulating the entire execution is impractical for the longer running benchmarks. The RPA-based profiles used for the co-location optimization are therefore collected by running the benchmarks on SPARC workstations. Obviously the SPARC processor does not support the RPA, so the RPA is emulated. The Strata compiler inserts instrumentation that emulates the RPA performing the co-location profiling task with the service threads including the adaptive sampling. The profiles produced are essentially identical to those that would be produced by the RPA.

Note that this step is only used to produce a co-location profile to drive the field selection algorithm. Profiled overhead is evaluated using SimpleMP simulations of the RPA, as described next.

6.9.2. Profile Overhead

The overhead for co-location profiling is measured while simulating the complete three-processor system-on-a-chip. The benchmark application runs on an 8-way out-of-order processor. Garbage collection runs on one service processor. The profile service threads run on the second processor. Two of these threads consume write-barrier profile records. The third service thread processes edge-profile records and co-location profile records.

One out of 256 branches are sampled for the edge profile. One out of 64 *getfields* are sampled for the co-location profile. Allocation sites and *putfields* are adaptively sampled as explained in Subsection 6.8.2, using sampling rates of one in one, one in eight and one in 64.

For each benchmark, the overhead is measured with three simulations of the same benchmark binary. The baseline simulation does not perform the co-location profiling. The RPA performs a reduced set of queries that only collect write-barrier and edge profiles. Overheads are computed by comparing execution times to this baseline simulation. The non-adaptive simulation adds co-location profiling, but adaptation is disabled. The final simulation performs co-location profiling with adaptive sampling.

The topology recognition algorithm is not yet fast enough to run efficiently on the service threads. Profile records are produced faster than they can be consumed. To stress the RPA, the profile overhead is taken with a simplified service thread that does keep up with the RPA.

6.10. Results Using the RPA-Based Profiler

This section evaluates the RPA-based profiler. The two major issues that need to be evaluated are the effectiveness of the profiling and the run-time overhead of the profiler.

Table 6.9. Fields co-located with RPA-based profiling, compared to ideal profiling

Benchmark	*Fields Co-Located*		*Changes Compared with Ideal*	
	RPA Profiler	*Ideal Profiler*	*Fields Added*	*Fields Removed*
	(1)	*(2)*	*(3)*	*(4)*
compress	8	8	*identical*	
db	1	1	*identical*	
espresso	11	9	2	0
jack	13	13	*identical*	
java_cup	11	16	0	5
javac	7	8	1	2
jess	19	20	1	2
jlex	8	9	1	2
mpegaudio	32	32	3	3
raytrace	9	9	1	1
rpa	12	17	0	5
strata	28	27	1	0
xp	5	4	1	0

6.10.1. Fields Co-Located

Table 6.9 compares the fields co-located using the RPA-based profiler to the fields co-located by the ideal profiler. Column 1 shows the number of fields co-located by the RPA-based profiler. Column 2 lists the number of fields selected by the RPA-based profiler that were not selected by the ideal profiler. Column 3 lists the number of fields not selected by the RPA-based profiler that were selected by the ideal profiler.

In several cases, *compress*, *db* and *jack*, exactly the same fields are selected. Most other benchmarks differ only by two or three fields. Furthermore, the changes are mostly due to minor differences in the importance of the fields. Step 6 of Figure 6.12 removes the least significant fields, based on a 1% threshold. Slight changes in the estimated improvement of fields, calculated by Equation 6.1, causes fields to slip back and forth across this threshold.

The RPA-based profiler does miss fields in some cases. Two *getfields* for a single object must be profiled to evaluate the stability of a field because the stability of the field is evaluated across multiple accesses. Given a sampling rate of one *getfield* in 64, fields that are accessed around 128 times per allocated object may not be profiled often enough for the profiler to observe. This tends to be more of a problem in the shorter running benchmarks, and such fields tend to be on the borderline of what should be co-located.

6.10.2. Performance of Co-Location Using the RPA-Based Profiler

Given the similarity of the co-location commands, it is not surprising that the performance of the co-location optimization is also similar. Column 1 and 2 of Table 6.10 shows the percent change in execution time between co-location driven by ideal profiling, and co-location driven by RPA-based profiling. Column 1 shows these results for co-location without optimized *getfields*. Column 2 shows the same results with optimized *getfields*.

In most cases the differences are small. When exactly the same fields are selected for co-location the results are identical. When *getfields* are not optimized, the differences are all less than 0.5%. When *getfields* are optimized, two benchmarks run significantly better when driven by the RPA-based profiles. *raytrace* gains an additional 1.3% and *xp* gains an additional 0.7%. A third benchmark, *jlex*, loses 1.5%.

Column 3 shows the final reduction in execution time achieved by speculative object co-location, using the RPA-based profiler. As the prior two columns imply, the results are not significantly different from previous results based on the ideal profiler.

6.10.3. Profiling Overhead of the RPA-Based Profiler

Figure 6.27 and Figure 6.28 highlight the benefits of adaptive sampling. Figure 6.27 shows the profiling overhead for the non-adaptive profiler. Benchmarks typically run at half the speed; *jack* with the unfortunately frequent reference store, runs ten times slower.

Fortunately, adaptive sampling corrects this situation in most cases (Figure 6.28). Seven

Table 6.10. Performance of co-location, driven by RPA-based profiling.

Benchmark	*Interval*	*% Change in Execution Time*		
		Relative to Ideal		*Relative to*
		Mem-only	*Total*	*Base*
		(1)	*(2)*	*(3)*
compress	1	*identical*		1.16
compress	2	*identical*		2.09
compress	3	*identical*		-2.39
compress	tot	*identical*		0.80
db	1	*identical*		9.66
db	2	*identical*		5.65
db	tot	*identical*		8.04
jack	1	-0.01	-0.00	-0.40
jack	2	0.01	-0.01	0.31
jack	3	-0.03	-0.07	0.19
jack	tot	-0.01	-0.02	0.02
jlex	tot	0.09	-1.45	1.28
mpegaudio	tot	-0.15	0.17	0.46
raytrace	tot	0.38	1.34	3.10
javac	1	-0.31	-0.10	-1.07
javac	2	-0.15	-0.40	-0.69
javac	tot	-0.23	-0.25	-0.88
xp	tot	0.10	0.72	-1.04
espresso	tot	-0.48	0.29	0.28
java_cup	tot	0.17	-0.20	-0.46
jess	tot	*identical*		0.50
strata	1	-0.06	-0.67	0.66
strata	2	-0.02	-0.02	0.71
strata	3	-0.06	0.06	0.74
strata	tot	-0.05	-0.18	0.71
rpa	tot	-0.00	-0.49	-1.20

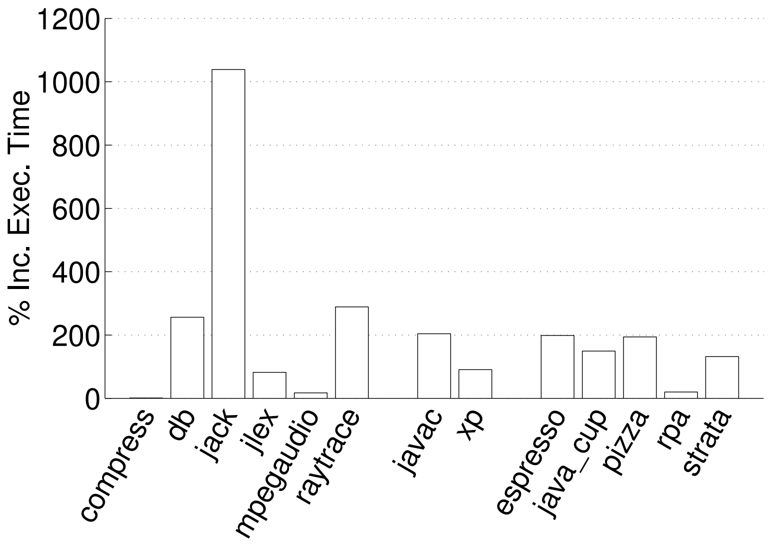


Figure 6.27. Profiling overhead without adaptive sampling

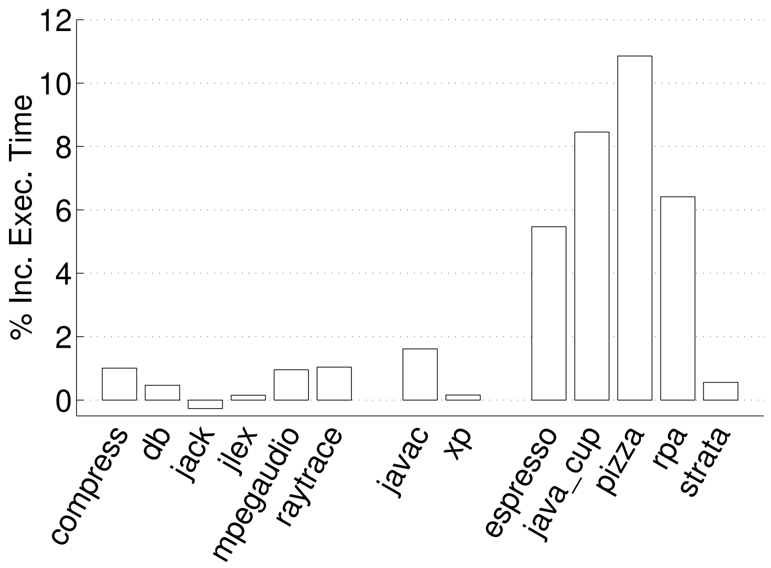


Figure 6.28. Profiling overhead with adaptive sampling

Table 6.11. *getfield* profiling rate and processing rate.

Benchmark	*Average Record Processing Time (Cycles) (1)*	*Average Time Between Records (Cycles) (2)*
compress	1197	668
db	1340	1039
jack	1294	3030
jlex	1378	2046
mpegaudio	1330	2333
raytrace	1289	1029
javac	1286	1824
xp	1365	2099
espresso	1288	4786
java_cup	1188	12397
jess	1361	1303
rpa	1331	15198
strata	1351	3891

benchmarks show overheads of less than 1%. Two more have an overhead less than 2%.

The remaining four benchmarks have higher overheads, from 5.5% to 10.9%. The benchmarks are all short, executing between 18.6 million instructions (*rpa*) and 81.5 millions instructions (*espresso*). Close observations of the behavior of individual profiled instructions indicate that the service threads are falling behind when encountering many never-before-seen instructions. The service threads use a hash table to map instruction PCs to execution counts. Inserts of new instructions into this table are currently not fast enough to keep up with application execution in some cases.

6.10.4. Profile Algorithm Execution Overhead

The previous results shows profile overheads with a simplified service thread that did not perform topology recognition. This is because the topology recognition algorithm is not fast enough to run on-line using a single service thread. Table 6.11 compares the average time, in cycles, to process a single record from a *getfield*, and the time between profiling *getfields*. In several cases *getfields* are executed faster than they can be processed, which would lead to high profiling overheads. It seems likely, however, that further effort can improve the performance of the topology recognition algorithm.

6.11. Related Work

Data-layout optimizations can be divided into three categories. Object inlining fuses multiple separate objects into a single object. Object co-location moves objects near each other to improve cache locality. Data restructuring includes a variety of techniques for improving cache locality by restructuring individual objects and arrays.

6.11.1. Object Inlining

Object inlining truly fuses multiple separate objects into one. This may involve eliminating object headers, and redundant reference fields. As a result, code may be extensively modified. For instance, references to the inlined object may no longer be valid. Such references will have to be changed to the container object, and accesses updated to reflect new field offsets. Dolby et al. [49, 50, 48] have been investigating powerful compiler-driven forms of object inlining for a number of years.

Although complex, this form of optimization has many potential benefits. It can improve cache behavior, both by shrinking the size of objects and by allocating together objects that are accessed together. Pointer chasing between the inlined objects is completely eliminated. Alias analysis may be improved, making other optimizations, such as register promotion (PRE of loads) more effective. Finally, it can reduce allocation costs, since multiple objects are allocated together.

The object inlining system developed by Dolby et al. is very general. The work is based on the Illinois Concert System [29], a research compilation framework for a type-safe dialect of C++. The analysis is an interprocedural whole-program analysis, but uses an adaptive form of analysis to reduce analysis costs. Static analysis finds *one-to-one fields*, a topology essentially equivalent to Figure 6.9a.

The result is a quite powerful system. For instance, unlike this work, array elements can be inlined into arrays. On average over a suite of data-structure intensive programs, object inlining eliminated 35% of the bytes allocated, 58% of the objects allocated. Reads of object fields are reduced by 28% and writes by 15%. Overall execution time improves by 14% on average.

Compared with object inlining, this work simplifies the transformation using two approaches. First, the format of neither container objects nor co-located objects is altered in complex ways. This allows co-located objects to be referenced directly by other objects without complication. Second, static analysis is replaced with dynamic verification, which ensures correct execution even when the co-location optimization fails.

This has two primary benefits. First, the analysis and correctness requirements are much simpler, a possible advantage during development. Second, the optimization can be applied even where static analysis would fail. This includes situations where the analysis is not robust enough to determine the invariants, and situations where the invariants actually do not hold. As the results in Section 6.7 will show, some objects are co-located, even though the co-location transformation fails in some cases.

Object inlining has also been pursued in other less general ways. In conjunction with functional programming languages, *unboxing* produces customized layouts to eliminate frequent pointer dereferencing [123, 74]. Similarly, Shao et al. focus on lists in functional languages by storing multiple data items in each link [112].

6.11.2. Object Co-Location

Rubin et al. [112] develop a C library for linked lists using an approach similar to Shao. Between eight and ten consecutive list nodes are kept in a *virtual cache line*. When combined with prefetching, primitive list operations, particularly traversal, are accelerated by as much as a factor of three over a naive linked-list implementation.

More general object co-location is typified by the work of Chilimbi and Larus [33]. Objects are located near each other to improve cache locality. Object structure is not modified. Chilimbi and Larus take advantage of a copying garbage collector to do this dynamically for several Cecil programs. A central contribution is a low-overhead (3% to 5%) profiling scheme based on instrumentation. The instrumentation builds a continuous trace of accessed object addresses, taking advantage of multiple accesses to the same object to reduce instrumentation overhead. Prior to garbage collection this list is traversed to build an *object affinity graph*. This graph indicates which objects are accessed together, and is used to guide the copying process.

Note that it would be trivial to collect the address information using the RPA. A service thread could build the address trace or the object affinity graph on the fly.

Chilimbi et al.[32] extend object co-location to C code using two semi-automatic mechanisms. The first, *ccmorph*, is a library that reorganizes tree-like data structures by clustering small subtrees onto cache lines. The programmer inserts calls to *ccmorph*, supplying a pointer to the root of the tree. The *ccmorph* routine shuffles the nodes around in memory to improve cache locality. Misuse of *ccmorph* will break a program. In particular, the programmer must ensure that there are no pointers into internal nodes of the tree, as *ccmorph* will move the nodes in memory. For speed and flexibility *ccmorph* is a template-based system. Reorganization code is automatically generated from a structural description of the tree.

The second mechanism, *ccmalloc*, is a modified version of the *malloc* function. The programmer specifies a pointer to another block that is thought to be accessed contemporaneously with the newly allocated block. *Ccmalloc* attempts to allocate the new block on the same cache line with the given block. Since the given block is only a hint, *ccmalloc* is safe. Misuse affects performance, but not correctness.

Seidl and Zorn [110] also develop an allocation mechanism for C code to improve spatial locality. They use a trace containing object allocations, deallocations and accesses to categorize objects as highly referenced, not highly referenced, short-lived or "other" Based on this separation, their system produces customized allocation code that predicts categories for objects at run-time. The custom allocator places objects from each group into separate 2MB memory segments. Results for two predictors based on the calling context at the time of allocation are presented.

The cache conscious data placement strategy developed by Calder et al. [25] seeks to improve the cache performance of stack, global variable and heap accesses by reducing conflict misses and improving spatial locality. A program profile is used to develop a *Temporal Relationship Graph* (TRG). Weighted arcs between data objects in the TRG, global variables, the stack, and heap allocated blocks, indicate how often data objects are accessed contemporaneously. Using this information, estimates of the number of cache misses can be made for different cache alignments. A greedy algorithm uses these estimates to select favorable starting addresses for each data object.

Rearrangement of global variables and the stack is fairly straightforward, since these are decided at link time and program startup. Controlling the layout of heap objects is more complex. Similar to the work of Seidl and Zorn [110], the system produces custom allocators that use the calling context to identify which data objects should be placed carefully. At run-time, the modified allocation routine places such objects together in the heap.

Luk and Mowry [88] suggest *memory forwarding* as a mechanism to enable aggressive relocation of data objects. A one-bit tag is associated with each eight bytes in memory to indicate if the double word contains a forwarding pointer. Upon accessing a tagged double word, hardware redirects the access based on the forwarding pointer (or invokes an interrupt that performs the relocation). The scheme allows aggressive data relocation offering substantial speedups, and is almost completely safe for arbitrary C code. Perhaps the greatest limitation is that memory vacated by relocated data is filled with forwarding pointers, and can never be reused.

6.11.3. Data Restructuring

Methods for restructuring arrays have been extensively studied, particularly for numerical FORTRAN programs. These techniques typically focus on kernels of nested loops, and seek to change the way arrays are laid out in memory to improve cache locality and reduce false sharing patterns in multiprocessor workloads [73, 7, 120, 81]. Arrays may be restructured through dimension reording, blocking (or tiling), and strip-mining [2] [7].

Chilimbi et al. [31] study a method of reorganizing and splitting Java objects for improved cache locality. Profile information determines which fields are accessed frequently (*hot fields*) and which are not (*cold fields*). Heuristics split classes into hot and cold portions, which are allocated separately with a reference from the hot portion to the cold portion. Following this transformation, object co-location [33] naturally moves the hot portions together. The entire process is completely automated and transparent, due to Java's object-oriented model.

The same work also explores the reordering of fields in C code. C is chosen because structures are often larger; reordering alone is unlikely to be beneficial for objects smaller than a cache line. A memory access trace, along with static program analysis, is used to determine which fields are accessed contemporaneously. Heuristics attempt to place fields

[2] Not to be confused with the loop transformation of the same name.

these together. This tool, *bbcache*, is semi-automated. Since changing field order in C can affect correctness *bbcache* makes only recommendations to be implemented by a programmer.

Truong et al. [126] also study reordering fields and interleaving C structures. The reasoning is similar to the work just described [31], though the technique is not automated. Fields in C structures are reordered, separating the structure into hot and cold portions. Since most structures are smaller than a cache line, this by itself is not effective. Hence, multiple instances of the structure are *interleaved*. This packs the hot fields of many instances together in one region, with the cold fields in another, a large constant offset away. This is accomplished by modifying the C structure declaration by hand. Fields are reordered, and padding is inserted to separate the hot and cold portions. Special allocation functions, *ialloc* and *ifree*, handle the interleaved allocation. Several limitations arise because the technique is applied at the C source level. For instance, arrays of transformed structures cannot be allocated efficiently because the padding space would be wasted.

Kistler and Franz [77] also describe a technique for reordering fields within objects. Their implementation, unlike any other presented here, is a completely transparent optimization performed at run-time by a virtual machine. This is a VM for the type-safe language Oberon based on Slim Binaries [76, 58]. Like Calder et al. above [25] they build a temporal relationship graph, upon which they base field ordering decisions. Unlike that work, they build this graph from path profiles, inferring the ordering accesses from program control flow. This likely yields much lower overheads than the trace-based methods used by other implementations. Fields are first reordered between cache lines, to improve spatial locality, and then reordered within a cache line to take advantage of the order in which the hardware services the words following a cache miss. While they show significant speedups for several benchmarks, the cost of path profiling is as much as 15%, and results in a performance loss in one case. Clearly this is an area where the RPA could provide additional benefit.

Note that many of the techniques described can be combined, likely with good effect. For instance, it may be reasonable to apply object inlining [49], producing larger objects that can be reorganized and split into hot and cold portions [31]. Finally, object co-location [33, 110] can be applied to improve cache locality or virtual memory system performance.

To this end, Rubin et al. [107] describe a unified evaluation framework for a wide range of data-layout optimizations such as those just discussed. Their system provides a single approach for evaluating which of several optimizations may be beneficial, which objects they should be applied to, and how. For instance, when applied to field reording, their technique would evaluate which structures should be rearranged, and which ordering is optimal. The technique is based on trace-based cache simulation of *data-objects trace*. For each access, the data-objects trace contains a semantic name for the accessed value in addition to the memory address. The trace is compressed by representing it as a context-free grammar [30]. This both reduces the storage for the trace, and accelerates the cache simulation in several novel ways. Given an efficient evaluation mechanism, the system evaluates hundreds or thousands of possible optimization configurations, and selects the best configuration.

6.12. Summary

An initial implementation of speculative object co-location improves the performance of six benchmarks by an average of 2.7%. These benchmarks would only improve by 1.8% on average, were *getfields* not optimized to take advantage of the co-location. These results indicate that pointer-chasing overhead should be considered as a potential source of speed-up, in addition to increasing spatial locality.

Further study is required for speculative object co-location to be a practical optimization. Potential gains can be made through careful interaction with the global register allocator, and through reducing the overhead of the speculative checks. It seems likely that the checks could be performed at the rarely executed *putfields*, rather than the frequently executed *getfields*. In addition, several common code idioms have been identified, which may provide direction for future work.

The RPA performs admirably as a source of profile information for object co-location. The profiles produced by the RPA are essentially the same as those produced by an intrusive ideal profiler. In this study, the RPA simultaneously performs five different profiling tasks, three for co-location profiles, one for concurrent garbage collection, and one for edge profiling.

Two of these tasks have different profiling requirements for the same instructions. The object co-location profiler has to collect PCs of store instructions with an adaptive random sampling rate. The concurrent GC algorithm has to collect the input values for the same stores, but without random sampling. The RPA performs both tasks simultaneously, providing evidence of its flexibility. Multiple profiling tasks for different purposes can be combined, even when the tasks have different profiling requirements for the same instructions. This will aid VM development, by providing a degree of isolation between profiling tasks developed independently within a large VM software development project.

Co-location profiles are collected with less than 3% overhead for ten of the 13 benchmarks. These low overheads are achieved through adaptive sampling. The profile tags provided by the RPA allow the co-location profiler to control the sampling rate on a per-instruction basis. High sampling rates provide accurate estimates for rarely executed instructions, which are essential to the co-location field selection algorithm. Low sampling rates reduce the profiling overhead for frequently executed instructions.

The remaining four benchmarks have higher overhead, 5.5% to 10.9%, because their short execution time do not allow the profiler to adapt. Improvements in the service thread code are needed if short-running applications must be supported. Note, however, that these issues are algorithmic, and are not related to the RPA itself.

CHAPTER 7
SUMMARY AND CONCLUSIONS

This dissertation develops and evaluates the relational profiling architecture (RPA). The RPA extends an implementation ISA with a rich set of powerful primitives for profiling and instrumentation designed to support service threads within a multithreaded virtual machine. These features focus on intelligent selection of information and efficient communication of this information to service threads.

The RPA is evaluated along three dimensions—expressiveness, implementation and performance. Expressiveness is evaluated using two novel applications for the RPA, concurrent garbage collection and speculative object co-location. These two applications demonstrate several characteristics of the RPA. An implementation is described that forms the basis for cost estimation and cycle-level simulation. Performance simulations evaluate the profiling overhead exhibited by these two applications.

7.1. Architectural Features

The architectural features of the RPA focus on intelligent selection of information and efficient communication of this information to service threads. Intelligent selection of information reduces the profile bandwidth seen by the service threads and simplifies profile processing. Service threads receive exactly what they need, and only what they need. An efficient communication mechanism eliminates disruptive application interrupts, drastically reducing profiling overhead.

The RPA expresses profiling and instrumentation tasks using queries. Queries embody the following four architectural features.

7.1.1. Intelligent Instruction Selection

The RPA selects instructions based on instruction op-codes and a dedicated profile tag in each instruction. Instruction op-codes are divided into eight op-code classes. Each class is further subdivided based on the two profile tag bits, yielding a total of 32 profile classes. For each profile class, the information collected, the sampling rate and Boolean pattern matching can independently configured.

7.1.2. Instruction Instrumentation

The RPA has the ability to guarantee that information is collected from all executions of instructions in a particular class. This allows the RPA to be used for instrumentation, to collect information that is required for correctness.

7.1.3. Boolean Pattern Matching

Sampled instructions produce a profile record containing the desired information. Queries can evaluate these records for interesting properties using arbitrary Boolean expressions built using if-then-else decision trees. Queries that are not of interest are typically discarded.

7.1.4. Shared-Memory Message Passing

Records that are of interest are typically sent to service threads through shared memory using FIFO queues. The service threads read these records using normal loads and stores.

The RPA can manage multiple queues, either for multiple types of messages or for added processing bandwidth. For added processing bandwidth, the RPA arranges multiple message queues into a *pool*. Pools allow multiple service threads, one per message queue, to work together to process a high-bandwidth stream of profile records. The RPA performs load balancing among the queues, sending new records to the queue that is most nearly empty.

Table 7.1. RPA component costs.

Component	*Size*
Profile Control Table	175 bytes of SRAM, LSFRs
Profile Networks	4 networks
Profile Buffers	8 buffers, 256 bytes of SRAM
Query Engine	Four stage simple pipeline
Message Engine	14 bytes of SRAM per queue

Profile records have a header that identifies the source of the record. The service threads use this header to distinguish between records of different types. This allows a single service thread to process multiple record types sent to the same message queue.

7.2. Implementation

These expressive features do not come at a high cost. This dissertation explores one particular design, demonstrating that the RPA can be practically implemented.

Table 7.1, duplicated from Section 4.2, shows the various hardware components that make up the example implementation of the RPA. The profile control table (PCT) contains a configuration for each of the 32 instruction classes. During dispatch, the PCT selects instructions to be profiled.

As the selected instructions flow through the pipeline, information is collected and sent to profile buffers via a set of profile networks. Simulations indicate that four networks and eight buffers sufficiently reduce dispatch stalls due to limited hardware resources. The profile network is latency insensitive. Latencies of at least 20 cycles can be tolerated and extra profile buffers can be used to compensate for network latency. However, the profile networks are a critical resource that a real implementation should attempt to optimize.

The query engine performs Boolean pattern matching, and sends profile messages via the message engine. The query engine is a simple four-stage pipeline capable of processing up to four queries simultaneously. As the profile buffers fill, the processing power of the query engine increases to one query instruction per cycle, providing optimum performance when it is needed.

7.3. Applications of the RPA

Two novel application of the RPA demonstrate its usefulness. The RPA enables a concurrent garbage collector to supply this object-oriented feature with almost no overhead. The RPA collects high-quality profiles for speculative object co-location with little overhead in most cases.

7.3.1. Concurrent Garbage Collection

The concurrent garbage collector requires a write barrier to examine all writes of references to objects in the heap. Previous implementations of the write barrier included write-barrier instrumentation code in the application and specialized hardware support. Using the RPA and service threads produces a write-barrier implementation combining the flexibility of software and the performance of dedicated hardware.

The resulting concurrent generational GC algorithm has overheads less than 0.3% typically and 3.7% in the worst case. Application pauses, a common problem with other GC algorithms, are never longer than 55,000 cycles or 55μS on a 1GHz processor.

To accomplish this feat, the RPA instruments selected store instructions and passes the needed information to service threads. Using this information the service threads perform the write barrier required by the garbage collector for correctness.

7.3.2. Speculative Object Co-location

An initial implementation of speculative object co-location improves the performance of six of 13 benchmarks by an average of 2.7%. However, further study is required for speculative object co-location to be a practical optimization. Potential gains can be made through careful interaction with the global register allocator, and through reducing the overhead of the speculative checks.

The RPA is an excellent source of profile information for object co-location. In this study, the RPA simultaneously performs five different profiling tasks, three for co-location profiles, one for concurrent garbage collection, and one for edge profiling. The RPA produces essentially the same quality profiles as an intrusive ideal profiler.

Co-location profiles are collected with less than 3% overhead for ten of the 13 benchmarks. These low overheads are achieved through adaptive sampling. High sampling rates provide accurate estimates for rarely executed instructions, which are essential to the co-location field selection algorithm. Low sampling rates reduce the profiling overhead for frequently executed instructions.

The remaining four benchmarks have higher overhead, 5.5% to 10.9%, because their short execution time does not allow the profiler to adapt. This indicates that improvements in the adaptation rate are needed if short-running applications must be supported. Note, however, that these issues are algorithmic, and are not related to the RPA itself.

7.4. Architectural Benefits

The profiling applications just described derive substantial benefit from the core RPA features.

7.4.1. Message Passing

Sending profile records through shared memory is an essential part of both applications. The interrupt-based communication mechanism used by many profilers would have been inadequate for these tasks.

The message passing mechanism supports efficient service thread design. Service threads executed only a 2.5 instructions of overhead per message. The write-barrier service threads used multiple message queues to easily provide additional record processing power. The co-location profiler used the header word to extract multiple message types from a single queue. Furthermore, simulations indicated that the RPA messaging used less than 9% of the shared processor interconnect.

7.4.2. Per-Instruction Profile Tag

Both applications of the RPA use the profile tag. The garbage collector requires that information be collected from certain store instructions. The profile tag identified these store instructions. Profiling all stores, rather than just relevant ones, is prohibitive, and complicates the service thread's function.

Speculative object co-location uses the profile tags to implement adaptive sampling. The profile tag encodes a sampling rate, which the service thread manipulates on a per-instruction basis.

Furthermore, initial evidence indicates that two bits is about the right size for the profile tag. One bit would not be enough to perform adaptive sampling, which likely requires at least three states (e.g. "don't sample", "sample frequently", "sample rarely").

7.4.3. Instruction Instrumentation

Traditional profiling tasks collect information only for use as a performance hint. Hence, random sampling is useful to provide a statistically reliable estimate of program behavior. The concurrent GC profiling task is unusual in that all information is required for correctness. The RPA's provision for guaranteed instrumentation is therefore essential. It is expected that this capability will open up other interesting new profiling applications.

7.4.4. Boolean Pattern Matching

The Boolean pattern matching capabilities of query clauses are used by both profiling applications. The concurrent GC query uses this capability to filter writes of null references. This simplifies the write-barrier code by eliminating a null-reference check. It also reduces the number of record processed by the write-barrier service threads. If it were not for this capability, one benchmark, *jack*, would completely overwhelm the service threads with a barrage of null-reference stores.

The co-location profiling task uses this capability to down-sample records from the same store instructions that are profiled for GC. The co-location profiling task randomly samples these stores to reduce the profile stream bandwidth. The RPA profiles all such stores, sending the required records to the GC service threads. The query engine randomly filters this stream, sending a small portion of these records to the co-location profile service thread.

7.4.5. Composing Profiling Tasks

The architectural features of the RPA make *composing* profiling tasks easy. Different profiling tasks can be combined and performed simultaneously, even when the tasks have contradictory requirements for the same instructions.

This was demonstrated when the GC profiling task was combined with the co-location profiling task. Both tasks require different data at different rates from the same set of store instructions.

The RPA allows both sets of data to be collected, because RPA queries explicitly encode which data to collect. The requirements are combined by taking the union of the information needed. The RPA allows the same instruction to be sample at multiple rates by initially sampling at the higher rate, and then down-sampling using the query engine, as just explained above.

This is likely an important characteristic of the RPA. In a large VM development project, multiple profiling tasks are liable to be constructed simultaneously by different parts of the development team. Supporting composition increases the likelihood that independently developed profiling tasks can be combined.

7.5. Experience with Service Threads

Initial experience developing service threads for the RPA has led to some observations. Generally, the RPA is not the weak link in the process. Rather, the performance of the service thread(s) determines if a profiling task is practical. Fortunately, software performance is fairly malleable. When a service thread is not efficient enough, further software improvements can typically significantly improve performance.

The dead-lock scenario discussed in Subsection 4.3.1 is also an important discovery. This scenario is very general, practically preventing service threads from using basic system services such as dynamic memory allocation. Furthermore, this deadlock case appears to apply to almost any profile system, not just RPA-driven service threads.

Dropping some randomly selected samples cures this ill, greatly simplifying the development of sophisticated service thread code. Randomly selected samples are dropped if they cause the processor to stall for a long period of time. This solution is only viable with message-passing-based profiling. It is not be feasible with interrupt-driven approaches, because the interrupt invokes the profile code before a problem can be detected.

7.6. Further Research

This research demonstrates that the RPA provides a powerful set of primitives that VM software can use to build sophisticated systems for information collection and analysis. But this is not the end. This research has just begun to show potential. Future research directions are briefly discussed below.

7.6.1. Further Architectural Extensions

Two extensions to the RPA seem of particular value and interest. While the RPA is good at collecting information related to a particular instruction, collecting relationships between multiple instructions is problematic. Paired sampling implemented by ProfileMe [41] may provide a good solution. Paired sampling profiles pairs of dynamically close instruction. Paired sampling could be generalized under the RPA to be quite powerful. For instance, sampling an instruction from one profile class (op-code and profile tag) could trigger sampling of successive instructions in another class. Such a system may have the ability to express queries such as, "Sample store instructions, and some following loads following selected stores."

It may also be beneficial to build profile records explicitly using special instructions. Additional instructions could explicitly allocate and fill profile buffers. This feature could combine information from more than one instruction, and may reduce the demand on the profile networks.

7.6.2. Further Profiling Applications

Applications like the GC algorithm, which requires the profile information for correctness, are of great interest. The RPA supports almost arbitrary producer-consumer communication between threads. This capability is likely useful in a wide range of standard libraries, for instance in container data-structures. Any time an activity can be delayed for some time, the RPA can be used to parallelize this activity. The RPA would send a request to service threads that perform the activity in parallel.

7.6.3. Automatic Composition of Queries

The GC profiling task and the co-location profiling task were combined by hand. Automatic composition may be possible. This could be useful as an off-line development tool to simplify the integration of multiple profiling tasks. Even more interesting, the system could run on-line, dynamic composing profile tasks presented at run-time.

BIBLIOGRAPHY

[1] Pentium(r) pro family developer's manual, volume 3: Operating system writer's manual, Dec. 1995. Intel Order Number 242692.

[2] Extended library functions cpc(3cpc), Feb. 2000.

[3] A. Agarwal, B. H. Lim, D. Kranz, , and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the* 17^{th} *Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[4] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 226–236. IEEE Computer Society Press, Nov. 1998.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, , and J. Whaley. The jalapeno virtual machine. *The IBM Systems Journal*, 39(1):211, Feb. 2000.

[6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *1990 International Conference on Supercomputing*, pages 1–6, 1990.

[7] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the* 5^{th} *ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 166–178. ACM Press, Aug. 1995.

[8] 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.

[9] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

[10] E. Armstrong. Hotspot, a new breed of virtual machine. *JavaWorld*, Mar. 1998.

[11] K. Arnold and J. Gosling. *The Java programming language (2nd ed.)*. ACM Press/Addison-Wesley Publishing Co., 1998.

[12] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F.Sweeny. Adaptvie Optimization in the Jalapeno JVM. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications*, Oct. 2000.

[13] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 134–145, June 1997.

[14] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 92–103, June 2001.

[15] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, Apr. 1978.

[16] L. A. Barroso et al. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the* 27^{th} *Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[17] R. J. Bergeron. The performance of the nas hsps in3q93. Technical Report RND-009, Numerical Aerodynamic Simulation (NAS), NASA Ames Research Center, Aug. 1994. http://www.nas.nasa.gov/Pubs/TechReports/RNDreports/RND-94-009/RND-94-009h.html.

[18] V. Berstis. Security and protection of data in the IBM System/38. In *Proceedings of the* 7^{th} *Annual International Symposium on Computer Architecture*, pages 245–252, 1980.

[19] R. Bodik, R. Gupta, and V. Sarkar. Abcd: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 321–333. ACM Press, 2000.

[20] R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 1–14, June 1998.

[21] H. J. Boehm, A. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 257–264, June 1991.

[22] H. J. Boehm and M. Weiser. Garbage collection in an un-cooperative environment. *Software-Practice and Experience*, pages 807–820, Sept. 1988.

[23] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, , and S. R. Kunkel. A multithreaded

PowerPC processor for commercial servers. *IBM Journal of Research and Development*, 44(6):–, 2000.

[24] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, Dept. of Computer Science, Univ. of Wisconsin-Madison, 1997.

[25] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the 8^{th} International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149. ACM Press, Oct. 1998.

[26] 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.

[27] R. S. Chappel, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26^{th} Annual International Symposium on Computer Architecture*, pages 186–195, May 1999.

[28] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. Fx!32 - a profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March–April 1998.

[29] A. Chien, J. Dolby, B. Ganguly, V. Karamcheti, and X. Zhang. Supporting high level programming with high performance: The illinois concert system. In *Proceedings of the Second International Workshop on High-level Parallel Programming Models and Supportive Environments*, pages 15–24, June 1993.

[30] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN'01 conference on*

Programming language design and implementation, pages 191–202. ACM Press, Jan. 2001.

[31] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 13–24. ACM Press, May 1999.

[32] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 1–12. ACM Press, May 1999.

[33] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the international symposium on Memory management*, pages 37–48. ACM Press, Oct. 1998.

[34] Y. Chou and J. P. Shen. Instruction path coprocessors. In *Proceedings of the* 27^{th} *Annual International Symposium on Computer Architecture*, pages 270–281, June 2000.

[35] B. E. Clark. 64 bits, no buts – implementing a true 64-bit computer system. presentation, Mar. 1997.

[36] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of the* 28^{th} *Annual International Symposium on Computer Architecture*, pages 14–25. ACM Press, June 2001.

[37] T. M. Conte, K. N. Menezes, and M. A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In 29^{th} *Annual IEEE/ACM International Symposium on Microarchitecture*, pages 36–45, Dec. 1996.

[38] T. M. Conte, B. A. Patel, and J. S. Cox. Using branch handling hardware to support profile-driven optimization. In 27^{th} *Annual IEEE/ACM International Symposium on Microarchitecture*, pages 12–21, Nov. 1994.

[39] 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.

[40] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a message-driven processor. In *Proceedings of the* 14^{th} *Annual International Symposium on Computer Architecture*, pages 189–196, June 1987.

[41] 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. 1997. IEEE Computer Society.

[42] 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 30^{th} *Annual IEEE/ACM International Symposium on Microarchitecture*, pages 292–302, Dec. 1997.

[43] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the* 17^{th} *Annual ACM Symposium on Principles of Programming Languages*, pages 261–269, Jan. 1990.

[44] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++

programs. Technical Report CU-CS-665-93, Univ. of Colorado at Boulder, CS Department, 1993.

[45] K. Diefendorff. Compaq chooses smt for alpha. *Microprocessor Report*, 13(16), Dec. 1999.

[46] K. Diefendorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, 13(13), Oct. 1999.

[47] E. Dijkstra, L. Lamport, A. Martins, C. Scholten, and E. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, Nov. 1978.

[48] J. Dolby. Automatic inline allocation of objects. In *Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation*, pages 7–17. ACM Press, May 1997.

[49] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 345–357. ACM Press, May 2000.

[50] J. Dolby and A. A. Chien. An evaluation of automatic object inline allocation techniques. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 1–20. ACM Press, Oct. 1998.

[51] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the* 20^{th} *Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, Jan. 1993.

[52] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collec-

tor for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 274–284, May 2000.

[53] P. Dreyfus. Programming design features of the gamma 60 computer. In *Proceedings of the Eastern Joint Computer Conference*, pages 174–181, Dec. 1958.

[54] K. Ebcioglu, E. R. Altman, and E. Hokenek. A java processor based on fast dynamic vliw compilation. In *International Workshop On Security and Efficiency Aspects of Java*, Jan. 1997.

[55] K. Ebcioğlu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the* 24^{rd} *Annual International Symposium on Computer Architecture*, pages 26–37, 1997.

[56] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow analysis of branch mispredictions and its application to early resoulution of branch outcomes. In 31^{st} *Annual IEEE/ACM International Symposium on Microarchitecture*, pages 59–68, Nov. 1998.

[57] M. J. Flynn, A. Podvin, and K. Shimizu. A multiple instruction stream processor with shared resources. *Parallel Processor Systems, Technologies and Applications*, pages 251–286, 1970.

[58] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, Dec. 1997.

[59] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In 31^{st} *Annual IEEE/ACM International Symposium on Microarchitecture*, pages 173–181, Dec. 1998.

[60] J. G. E. Daddis and H. C. Torng. The concurrent execution of multiple instruction

streams on superscalar processors. In *International Conference on Parallel Processing*, volume I, pages 76–83, Aug. 1991.

[61] L. George and A. W. Appel. Iterated register coalescing. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 208–218. ACM Press, Jan. 1996.

[62] L. Geppert and T. S. Perry. Transmeta's magic show. *IEEE Spectrum*, 37(5):26–33, May 2000.

[63] J. Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall PTR, 2002.

[64] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. Dynamic and transparent binary translation. *Computer*, pages 54–59, Mar. 2000.

[65] R. Gupta and R. Bodik. Register pressure sensitive redundancy elimination. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 107–121, Mar. 1999.

[66] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999.

[67] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the* 19^{th} *Annual International Symposium on Computer Architecture*, pages 136–145. ACM Press, May 1992.

[68] U. Holzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Thesis CS-TR-94-1520, University of Stanford, Aug. 1994.

[69] U. Holzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, Department of Computer Science, Mar. 1995. Available as Sun Microsystems Laboratories Technical Report TR-95-35.

[70] R. J. Hookway and M. A. Herdeg. Digital fx!32: Combining emulation and binary translation. *Digital Technical Journal*, pages 3–12, Jan. 1997.

[71] L. Huelsbergen and P. Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In *International Symposium on Memory Management*, pages 166–175, 1998.

[72] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proceedings of the 5^{th} International Symposium on High Performance Computer Architecture*, pages 125–129, Jan. 1999.

[73] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, 1995.

[74] S. L. P. Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming and Computer Architecture*, Sept. 1991.

[75] E. F. M. Jr. A multiple stream registerless shared-resource processor. *IEEE Transactions on Computers*, 23(3):277–285, Mar. 1974.

[76] T. Kistler and M. Franz. Perpetual adaptation of software to hardware: An extensible architecture for providing code optimization as a central system service. Technical

Report TR 99–12, Department of of Information and Computer Science, University of California, Irvine, 1999.

[77] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):490–505, 2000.

[78] A. Klaiber. The technology behind Crusoe^{tm} processors, Jan. 2000.

[79] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

[80] S. M. Kurlander and C. N. Fischer. Minimum cost interprocedural register allocation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 230–241. ACM Press, Jan. 1996.

[81] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the* 4^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74. ACM Press, Apr. 1991.

[82] B. Lampson. Personal distributed computing: the alto and ethernet software. In *Proceedings of the ACM Conference on The history of personal workstations*, pages 101–131. ACM Press, 1986.

[83] M. Larose and M. Feeley. A compacting incremental collector and its performance in a production quality compiler. In *International Symposium on Memory Management*, pages 1–9, Oct. 1998.

[84] K. M. Lepak and M. H. Lipasti. On the Value Locality of Store Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[85] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *Proceedings of the 33rd annual IEEE/ACM international symposium on Microarchitecture*, pages 22–31. ACM Press, Dec. 2000.

[86] F. E. Levine. A programmer's view of performance monitoring in the powerpc microprocessor. *IBM Journal of Research and Development*, 41(3):345ff, 1997.

[87] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, USA, 1997.

[88] C. Luk and T. Mowry. Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation. In D. DeGroot, editor, *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, pages 88–99, May 1–5 1999.

[89] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), Feb. 2002.

[90] 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*, pages 136–147, June 1999.

[91] S. Microsystems. MAJC architecture tutorial. Technical report, Sun Microsystems, May 1999.

[92] S. Microsystems. The Java HotSpot (tm) Perfomance Engine: An In-Depth Look. Technical White paper, Apr. 2001.

[93] D. A. Moon. Garbage collection in a large LISP system. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 234–246, Aug. 1984.

[94] D. A. Moon. Symbolics architecture. *IEEE Computer*, 20(1):43–52, Jan. 1987.

[95] E. Morel and C. Renviose. Global optimization by supression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[96] P. A. Nelson. A comparison of pascal intermediate languages. In *Proceedings of the SIGPLAN symposium on Compiler construction*, pages 208–213, 1979.

[97] R. S. Ohran. *Lilith: A Workstation Computer for Modula-2*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 1984.

[98] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the hydra cmp. In *Proceedings of the 13th international conference on Supercomputing*, pages 21–30. ACM Press, June 1999.

[99] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7^{th} International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[100] J. O'Toole and S. Nettles. Concurrent replicating garbage collection. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 34–42, 1994.

[101] T. Parr. Another tool for language recognition (antlr), 2000. Available at http://www.antlr.org.

[102] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ilp processors. In *Proceedings of the 21ST annual international symposium on Computer architecture*, pages 120–129. IEEE Computer Society Press, 1994.

[103] R. G. Prasadh and C.-L. Wu. A benchmark evaluation of a multi-threaded risc processor architecture. In *International Conference on Parallel Processing*, volume I, pages 84–91, Aug. 1991.

[104] J. R. H. Halstead and T. Fujita. Masa: a multithreaded processor architecture for parallel symbolic computing. In *The 15th Annual International Symposium on Computer architecture*, pages 443–451. IEEE Computer Society Press, May 1988.

[105] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture*, pages 24–35. IEEE Computer Society Press, Dec. 1996.

[106] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pages 191–202, Jan. 2001.

[107] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *Conference Record of the* 29^{st} *Annual ACM Symposium on Principles of Programming Languages*, pages 140–153. ACM Press, Jan. 2002.

[108] 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.

[109] W. J. Schmidt and K. D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *Proceedings of the* 6^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–85, 1994.

[110] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the* 8^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23. ACM Press, Oct. 1998.

[111] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: a quasi-static compiler for java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 66–82, Oct. 2000.

[112] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 185–195. ACM Press, July 1994.

[113] B. J. Smith. Architecture and applications of the hep multiprocessor computer system. In *SPIE Real Time Signal Processing IV*, pages 241–248, 1981.

[114] J. E. Smith, T. Heil, S. Sastry, and T. Bezenek. Achieving high performance via co-designed virtual machines. In *International Workshop on Innovative Architecture*, pages 77–85, Maui, HI, USA, Oct. 1999. IEEE Computer Society.

[115] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425. ACM Press, May 1995.

[116] F. G. Soltis. *Inside the AS/400*. 29th Street Pr, Nov. 1997.

[117] Y. Song and M. Dubois. Assisted execution. Technical Report CENG 98-25, University of Southern California, Dept. of EE-Systems, Oct. 1998.

[118] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *The 2001 USENIX Annual Technical Conference*, June 2001.

[119] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 257–268. ACM Press, Nov. 2000.

[120] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *1993 International Conference on Supercomputing*, pages 410–419, Nov. 1993.

[121] C. Thacker. Personal distributed computing: the alto and ethernet hardware. In *Proceedings of the ACM Conference on The history of personal workstations*, pages 87–100. ACM Press, 1986.

[122] R. Thekkath and S. Eggers. The effectiveness of multiple hardware contexts. In *Proceedings of the* 6^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 328–337, Oct. 1994.

[123] P. J. Thiemann. Unboxed values and polymorphic typing revisited. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 24–35. ACM Press, 1995.

[124] M. R. Thistle and B. J. Smith. A processor architecture for horizon. In *1988 International Conference on Supercomputing*, pages 35–41, Nov. 1988.

[125] J. E. Thornton. Parallel operation in the control data 6600. In *American Federation of Information Processing Societies Fall Joint Computer Conference, Part II*, volume 26, pages 33–40, 1964.

[126] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 322–329, Oct. 1998.

[127] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, June 1995.

[128] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23^{rd} Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[129] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SIG-SOFT/SIGPLAN Practical Programming Environments Conf.*, pages 157–167, Apr. 1984.

[130] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. Architecture of soar: Smalltalk on a risc. In *Proceedings of the 11^{th} Annual International Symposium on Computer Architecture*, pages 188–197, June 1984.

[131] 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.

[132] D. W. Westcott and V. White. Instruction sampling instrumentation, Sept. 1992.

[133] C. Whitby-Strevens. The transputer. In *Proceedings of the 12th annual international symposium on Computer architecture*, pages 292–300. IEEE Computer Society Press, June 1985.

[134] P. R. Wilson. Uniprocessor garbage collection techniques. In *1992 SIGPLAN Intl. Workshop on Memory Management*, pages 1–42, Sept. 1992.

[135] M. J. Wing and G. P. D'Souza. Gated store buffer for an advanced microprocessor, Dec. 1996.

[136] N. Wirth. Lilith: A personal computer for the software engineer. In 5^{th} *International Conference on Software Engineering*, 1981.

[137] N. Wirth. From programming language design to computer construction. *Communications of the ACM*, 28(2):160–164, 1985.

[138] M. Wolczko and I. Williams. Multi-level garbage collection in a high-performance persistent object system. In 5^{th} *International Workshop on Persistent Object Systems*, Sept. 1992.

[139] M. Wolczko and I. Williams. The influence of the object-oriented language model on a supporting architecture. In 26^{th} *Hawaii International Conference on System Sciences*, pages 182–191, Jan. 1993.

[140] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

[141] W. Yamamoto, M. Serrano, A. Talcott, R. Wood, and M. Nemirovsky. Performance estimation of multistreamed, superscalar processors. In 27^{th} *Hawaii International Conference on System Sciences*, volume I, pages 105–204, Jan. 1994.

[142] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 124–134. ACM Press, Apr. 1992.

[143] C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith. Near-optimal intraprocedural branch alignment. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 183–193, June 1997.

[144] C. Zilles and G. Sohi. Execution-based Prediction Using Speculative Slices. In *Proceedings of the* 28^{th} *Annual International Symposium on Computer Architecture*, pages 2–13, July 2001.

[145] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 87–98, June 1990.

[146] B. Zorn. The measured cost of conservative garbage collection. Technical Report CU-CS-573-92, Univ. of Colorado at Boulder, CS Department, Feb. 1992.

[147] M. L. Zumbrunnen, A. H. Wottreng, S. R. Kunkel, , and J. M. Borkenhagen. V4r4 architecture, processor complex, and performance.