

Table of Contents

CHAPTER 1 Introduction	1
1.1 Contributions	2
1.2 Road Map	4
CHAPTER 2 Java and its Runtime Environment	7
2.1 Java Source Language	8
2.1.1 Short History of Java	8
2.1.2 Taxonomy of Java	10
2.1.2.1 Basic Aspects	10
2.1.2.2 Parallel Programming Support	11
2.1.2.3 Object-oriented Aspects	11
2.1.2.4 Programming in the Large	13
2.2 Java Standard Library	15
2.3 Java Runtime Environment	17
2.3.1 Components of the Runtime Environment	18
2.3.1.1 Abstract Processor	18
2.3.1.2 Abstract Memory Model	21
2.3.1.3 Continuous Background Services	23
2.3.2 Structure of Binary Classfiles	24
2.3.2.1 Representation of Defined Program Entities	24
2.3.2.2 Constant Pool	28
2.3.2.3 Concept of Attributes	30
2.3.3 Preparing Code for Execution	31
2.3.3.1 Loading	33
2.3.3.2 Verification	34
2.3.3.3 Preparation, Resolution, and Initialization	36
2.4 Java Features that Impact Optimization	38
2.4.1 Platform Independence	38
2.4.2 Dynamic Class Loading	40
2.4.2.1 Binary Compatibility	41
2.4.2.2 Reflection - a Limited Form of Metaprogramming	46
2.4.3 Garbage Collection and Multithreading	49
2.4.4 Access Control	51
2.4.4.1 Default Access (Package Scope)	51
2.4.4.2 protected Access	52
2.4.4.3 Restricting Access with final	54
CHAPTER 3 Static Analysis Furnishes Dynamic Optimization	57
3.1 Motivation	57

3.2	System Overview	60
3.2.1	Division of Tasks between Subsystems	61
3.2.2	Modelling of Incomplete Analysis Results	65
3.2.3	Design Criteria	66
3.2.4	Analysis of Incomplete Programs: Benefits and Consequences	67
3.3	Granularity of Analysis: Flexibility vs. Quality	72
3.3.1	Analysis of Complete Programs	72
3.3.2	Analysis of Individual Classes	74
3.3.3	Analysis of Complete Packages	75
3.3.4	Software Libraries as Groups of Related Packages	80
3.3.5	Implications of Dynamic Class Loading and Reflection	83
3.3.6	Integration with the Existing Tool Chain	84
3.4	Requirements on Software Libraries	86
3.4.1	Completeness of Packages and Sealing	86
3.4.2	Deployment Constraints	88
3.4.3	Grouping Packages into Libraries	90

CHAPTER 4 Implementation Techniques for Java Runtime Environments . 95

4.1	Standard Execution Engines	95
4.1.1	Simple Interpreters	96
4.1.2	Code-Rewriting Interpreters	97
4.1.3	Just-in-Time Compilers	99
4.1.3.1	Combining Different Execution Engines	102
4.1.3.2	Combining Differently Tuned JIT Compilers	104
4.2	State of the Art	105
4.2.1	Dynamic Analysis and Optimization of Java Programs	105
4.2.1.1	IBM JIT Compiler	106
4.2.1.2	Kaffe	108
4.2.1.3	HotSpot	109
4.2.1.4	Intel JIT Compiler	112
4.2.1.5	OpenJIT	114
4.2.1.6	Other Systems	115
4.2.2	Static Analysis and Optimization of Java Programs	117
4.2.2.1	Marmot	118
4.2.2.2	Toba	121
4.2.2.3	GNU GJ	122
4.2.2.4	Swift	123
4.2.2.5	Other Systems	124

CHAPTER 5 Program Analysis for Isolated Object-oriented Libraries 127

5.1	Formal Model	129
5.1.1	Analysis of Incomplete Programs	129
5.1.2	Method Families	134

5.1.2.1	Prerequisite Definitions	134
5.1.2.2	Characterization of Method Families	135
5.1.2.3	Definition of Method Families	139
5.1.2.4	Method Families Spanning Libraries.	143
5.1.3	Modelling Missing Parts of the Program	147
5.1.3.1	Categories of Dependency Structures for Program Entities	148
5.1.3.2	Capturing Partial Computations and Dependencies	151
5.1.3.3	Definition of Single Aspect Model	156
5.2	Modelling Interprocedural Analysis Problems	158
5.2.1	Properties of Methods.	159
5.2.1.1	Subexpression-preserving Methods.	159
5.2.1.2	Superfluous Constructors.	166
5.2.1.3	Escape Analysis.	169
5.2.2	Properties of Fields.	173
5.2.2.1	Immutable Fields	174
5.2.2.2	Non-null Fields	178
5.2.2.3	Type-restricted Fields	180
5.3	Deploying Single Aspect Models	181
5.3.1	Simplification of Single Aspect Models.	182
5.3.2	Describing Optimization Opportunities	184
5.3.3	Composition of Models at Runtime	185
5.3.4	Dynamic Class Loading	186
5.4	Related Work.	186
5.4.1	Analysis of Object-oriented Software	187
5.4.2	Jax.	189
5.4.3	Compositional Points-to and Escape Analysis	191
5.4.4	Field Analysis in the Swift Compiler	192
5.4.5	Static Analysis in spite of Dynamic Class Loading	194
5.4.6	Staged Optimization	196
5.4.6.1	Link time Optimization	198
5.4.6.2	Feedback-driven Optimization	199
5.4.6.3	Partial Evaluation and Runtime Code Generation.	201
5.4.6.4	Automatically Generated Annotations.	203
5.4.6.5	Manual Annotations.	205

CHAPTER 6 SAILDOWN: an Implementation of Partial Analysis 207

6.1	Static Analysis Phase.	208
6.1.1	Object Model for the Subject Software	210
6.1.2	Design Principles	212
6.1.2.1	Analysis Plug-ins	212
6.1.2.2	Transformation Plug-ins	218
6.1.3	Implementation of Partial Analysis Algorithms.	220
6.1.4	Auxiliary Analysis Techniques	222
6.1.4.1	Intraprocedural Analysis	222

6.1.4.2	Type Analysis	223
6.1.4.3	Dataflow Analysis in the Presence of Exceptions	224
6.1.4.4	Reflection	227
6.2	Augmented Runtime-Environment	228
6.2.1	Augmented JDK-1.2.2 JVM on Sparc	228
6.2.2	Organization and Contents of Classfile Attributes	231
CHAPTER 7 Performance Measurements		237
7.1	Analyzed Subject Software	237
7.1.1	Application Programs	238
7.1.2	Supporting Software Libraries	244
7.1.3	Software Interface between Applications and Libraries	247
7.2	Evaluation of Static Partial Analysis Phase	249
7.2.1	Structure of Analysis Results	249
7.2.1.1	Results for Subexpression-preserving Methods	250
7.2.1.2	Results for Superfluous Constructors	254
7.2.1.3	Results for Immutable Fields	256
7.2.1.4	Results for Non-null Fields	259
7.2.1.5	Results for Type-restricted Fields	262
7.2.1.6	Comparison of Analyses	265
7.2.2	Size of Library Annotations	270
7.2.3	Analysis Time and Space	272
7.3	Evaluation of Modified Runtime Environment	275
7.3.1	Dynamic Composition of Analysis Results	275
7.3.2	Dynamic Optimization	280
7.3.2.1	Results for Analyses Based on Methods	281
7.3.2.2	Results for Analyses Based on Fields	283
7.3.2.3	Comparison of Optimizations	286
7.3.3	Impact of Input Size	291
7.3.4	Contributions of Application Code and Library Code	294
7.3.5	Distribution of Dynamic Optimizations over Time	296
7.3.6	Speed Gains from Optimization	300
CHAPTER 8 Conclusion		305
8.1	Experiences	306
8.2	Future Improvements	307
Bibliography		311
Appendix		325
A.1	Proof for Distributivity of SAM Application over \oplus	325

List of Figures

CHAPTER 1 Introduction	1
Fig. 1-1 Static analysis of incomplete programs with dynamic composition	2
Fig. 1-2 Research topics related to hybrid program analysis and optimization	5
CHAPTER 2 Java and its Runtime Environment	7
Fig. 2-1 Translation and execution of a simple Java program.....	7
Fig. 2-2 Example inheritance relation with classes and interfaces	13
Fig. 2-3 Organizational structure of Java programs	14
Fig. 2-4 Collaboration between library, compiler and runtime environment	16
Fig. 2-5 Overview of the runtime environment.....	18
Fig. 2-6 Example Java method and translation to bytecode.....	20
Fig. 2-7 More detailed view of the memory model.....	22
Fig. 2-8 Elements of a classfile	25
Fig. 2-9 Examples for representation of program entities in classfiles	26
Fig. 2-10 Source code for a method and resulting representation in classfile.....	27
Fig. 2-11 Code transformation used to implement access to a class object	28
Fig. 2-12 Hierarchical dependencies among constant pool entries	29
Fig. 2-13 Preparing compiled classes for execution	32
Fig. 2-14 Merging operand stack type models	35
Fig. 2-15 Example exploiting resolution on demand	37
Fig. 2-16 JVM and standard library isolate code from platform differences	39
Fig. 2-17 Invocation of a newly added overriding method	42
Fig. 2-18 Method resolution in the compiler would hurt binary compatibility	44
Fig. 2-19 Example for a binary compatible change that is <i>not</i> source compatible	45
Fig. 2-20 Invoking a method via reflection	47
Fig. 2-21 Principal use of reflection for object serialization	48
Fig. 2-22 Java memory model obstructs caching of field values across reads	50
Fig. 2-23 Package scope and method overriding	53
CHAPTER 3 Static Analysis Furnishes Dynamic Optimization	57
Fig. 3-1 Compiling and deploying programs: C++ versus Java.....	58
Fig. 3-2 Separate analysis of software components and reuse of analysis results	59
Fig. 3-3 Analysis, sealing and execution of an incomplete piece of software	61
Fig. 3-4 Tasks performed by analysis phase	62
Fig. 3-5 Corresponding actions required from optimization phase.....	64
Fig. 3-6 Modelling partial results for a simple example analysis	66
Fig. 3-7 Deciding what static analyses and dynamic optimizations to perform	71
Fig. 3-8 C++ compiler and linker gradually analyze and seal a program	73
Fig. 3-9 Identifying common subexpressions in the context of a single class	75
Fig. 3-10 Visibility levels for declarations in the bytecode of the Swing library.....	76
Fig. 3-11 Visibility of instance variables and methods in different libraries	77
Fig. 3-12 Adding or changing classes invalidates analysis results.....	78
Fig. 3-13 Removal of a method introduces side-effects into its former callee.....	79

Fig. 3-14	Build process for a large Java program with libraries	80
Fig. 3-15	Flow of information during and after analysis of a group of packages	82
Fig. 3-16	Java archive tool augmented with static analysis	84
Fig. 3-17	Analysis information becomes invalid outside the archive	85
Fig. 3-18	Modifying a sealed library loses analysis information.....	87
Fig. 3-19	Examples for <i>unfaithful</i> class loaders.....	89
Fig. 3-20	Separate software libraries better preserve implementation traits.....	91
Fig. 3-21	Cyclic inheritance among different libraries	93

CHAPTER 4 Implementation Techniques for Java Runtime Environments... 95

Fig. 4-1	Steps of central interpreter loop	97
Fig. 4-2	Example for a code-rewrite on first execution.....	98
Fig. 4-3	Subtasks inside a JIT compiler.....	100
Fig. 4-4	Concept of optimistic optimization	110

CHAPTER 5 Program Analysis for Isolated Object-oriented Libraries 127

Fig. 5-1	Partial program available for analysis	132
Fig. 5-2	Example for a straightforward method family	136
Fig. 5-3	Method family without definition in root class	137
Fig. 5-4	Inheritance frontier for multiple inherited method declarations.....	138
Fig. 5-5	Bounding method families for package-scoped definitions	140
Fig. 5-6	Nested method families spawned by subclasses.....	141
Fig. 5-7	Non-minimal method family	142
Fig. 5-8	Method family spanning three libraries.....	144
Fig. 5-9	Example for a constrained method family.....	146
Fig. 5-10	Categories of program entities in relation to a software library	149
Fig. 5-11	Domain of a partial solution before and after simplification.....	156
Fig. 5-12	Method containing potential common subexpressions.....	160
Fig. 5-13	Range and entropy of PAS for subexpression-preserving methods.....	161
Fig. 5-14	Loop in method leads to cycle in SAM	165
Fig. 5-15	Examples for superfluous and useful constructors	167
Fig. 5-16	Example constellation for escape analysis	172
Fig. 5-17	Range, entropy, and binary operation of PAS for immutable fields	174
Fig. 5-18	Range and entropy of PAS and example for non-null fields	178
Fig. 5-19	Example for type-restricted fields	181
Fig. 5-20	Comparison of staged and annotation-based optimization.....	197
Fig. 5-21	Build process with feedback-driven optimizations.....	200

CHAPTER 6 SAILDOWN: an Implementation of Partial Analysis..... 207

Fig. 6-1	Static and dynamic subsystem of SAILDOWN.....	207
Fig. 6-2	Analyzing a library with dependencies on other libraries	208
Fig. 6-3	Software layers (libraries) in the SAILDOWN analysis phase.....	209
Fig. 6-4	Object model for the software to be analyzed	210
Fig. 6-5	Role of an analysis plug-in within the static analysis framework	213
Fig. 6-6	Graphical representations of analysis results	215
Fig. 6-7	Transformation plug-in as used to implement SAILDOWN	218
Fig. 6-8	Extensions to framework specific to partial analysis in SAILDOWN.....	221
Fig. 6-9	Usage of <code>try</code> blocks and exception handlers in the Swing 1.1.1 library.....	226

Fig. 6-10	SAILDOWN additions to stock JDK runtime environment	229
Fig. 6-11	Organization of analysis results in the custom classfile attributes	232
Fig. 6-12	Grammar for encoding of entity SAMs in custom classfile attributes	234

CHAPTER 7 Performance Measurements 237

Fig. 7-1	Screenshot impressions of <code>jEdit</code> (left) and <code>xmlviewer</code> (right)	239
Fig. 7-2	Library dependency structure of the subject applications	245
Fig. 7-3	Comparison of SAM structure for the five static partial analyses	266
Fig. 7-4	Comparison of indicator structure for the five static partial analyses	269
Fig. 7-5	Relative growth of classfiles caused by library annotations	271
Fig. 7-6	Successful dynamic optimizations for analyses based on methods	282
Fig. 7-7	Successful dynamic optimizations for analyses based on fields	285
Fig. 7-8	Successful dynamic optimizations for analyses based on fields (cont.)	286
Fig. 7-9	Comparison of dynamic optimizations based on the five analyses	287
Fig. 7-10	Effectiveness of dynamic optimizations based on the five analyses	288
Fig. 7-11	Impact of size of input on dynamic optimization of non-null fields	292
Fig. 7-12	Importance of interprocedural analysis for different sizes of input	294
Fig. 7-13	Dynamic optimizations in library code vs. application code	295
Fig. 7-14	Dynamic optimizations during a test run of <code>_213_javac</code>	297
Fig. 7-15	Dynamic optimizations during a test run of <code>jEdit</code>	299

CHAPTER 8 Conclusion..... 305

List of Tables

2-1	Access levels in Java.....	52
5-1	Example call sites and method results with matching SAM terms.....	164
5-2	Example constructors and matching constants for computation of SAM	176
6-1	Analysis plug-ins in our framework	217
7-1	Size of subject application programs	242
7-2	Static vs. dynamic occurrences of field accesses and method calls.....	243
7-3	Size of subject libraries.....	246
7-4	Static vs. dynamic software interface between applications and libraries.....	247
7-5	Structure of SAMs for subexpression-preserving methods	251
7-6	Structure of optimization indicators for subexpression-preserving methods ..	253
7-7	Structure of SAMs for superfluous constructors.....	255
7-8	Structure of optimization indicators for superfluous constructors.....	256
7-9	Structure of SAMs for immutable fields.....	257
7-10	Structure of optimization indicators for immutable fields.....	259
7-11	Structure of SAMs for non-null fields	260
7-12	Structure of optimization indicators for non-null fields.....	262
7-13	Structure of SAMs for type-restricted fields.....	264
7-14	Structure of optimization indicators for type-restricted fields	265
7-15	Time required to analyze pieces of subject software	273
7-16	Memory required to analyze pieces of subject software.....	274
7-17	Evaluation effort for two example analyses.....	276
7-18	Dependency tracking effort for two example analyses.....	278
7-19	Program characteristics that impact structural composition effort	279
7-20	Dynamic counts of envisioned optimizations for method analyses.....	281
7-21	Dynamic counts of envisioned optimizations for field analyses.....	284
7-22	Speed improvements under dynamic inlining optimization	301

List of Definitions and Theorems

Definition 3-1: Faithful class loader.....	88
Definition 3-2: Library inheritance graph	92
Definition 5-1: Method definition, implementation, declaration	134
Definition 5-2: Method characterization	135
Definition 5-3: Inheritance frontier	137
Definition 5-4: Method family	139
Definition 5-5: Method family for interface root type	141
Theorem 5-6: Completeness of method families.....	142
Definition 5-7: Constrained method family	143
Definition 5-8: Program entities.....	148
Definition 5-9: Analysis problem.....	152
Definition 5-10: Partial analysis system.....	153
Definition 5-11: Partial solution	155
Definition 5-12: Single aspect model.....	157
Definition 5-13: Normal form of SAMs.....	182
Theorem 5-14: Sufficiency of normal form	182
Theorem 5-15: Distributivity of SAM applications	183
Theorem 5-16: Monotony of SAMs	183