Modular Implementation of Program Adaptation with Existing Scientific Codes

Pilsung Kang

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Examining Committee:

Srinidhi Varadarajan, Chair
Naren Ramakrishnan, Co-Chair
Calvin J. Ribbens
Eli Tilevich
Danesh K. Tafti

August 10, 2010
Blacksburg, Virginia

Keywords: program adaptation, scientific computing, parallel programming

Modular Implementation of Program Adaptation with Existing Scientific Codes

Pilsung Kang

Abstract

Often times, scientific software needs to be adapted for different execution environments, problem sets, and available resources to ensure its efficiency and reliability. Directly modifying source code to implement adaptation is time-consuming, error-prone, and difficult to manage for today's complex software. This thesis studies modular approaches to implementing program adaptation with existing scientific codes, whereby application-specific adaptation strategies can be implemented in separate code which is then transparently combined with a given program. By using the approaches developed in this thesis, scientific programmers can focus on the design and implementation of adaptation schemes, manage an adaptation code separately from the main program components, and compose an adaptive application whose original capabilities are enhanced in diverse aspects such as application performance and stability. The primary objective of the modular approaches in this study is to provide a language-independent development method of adapting existing scientific software, so that applications written in different languages can be supported when implementing adaptation schemes. In particular, the emphasis is on Fortran, which has been a mainstream language for programming scientific applications.

Three research questions are formulated in this thesis, each of which aims to: design and implement high-level abstractions for expressing adaptation strategies, develop a dynamic tuning approach for parallel programs, and support flexible runtime adaptation schemes, respectively. The applicability of the proposed approaches is demonstrated through example applications to real-world scientific software.

// Acknowledgements

I would like to thank the people without whom this dissertation would not have been possible.

First of all, I would like to express sincere thanks to my advisor, Dr. Srinidhi Varadarajan, who guided me through the research in the parallel and high-performance computing areas with invaluable inspirations and supervision.

I am immensely grateful to Dr. Naren Ramakrishnan, who, as a co-advisor, guided me to survive during hard times with warm encouragement and advice. He taught me how to design and perform research in computer science.

I would like to thank Dr. Calvin J. Ribbens for his extremely valuable suggestions and feedback on the subjects of scientific computing and parallel computing. Besides, I also learned a lot about writing a research paper through collaborations with him.

Dr. Danesh K. Tafti offered various adaptation scenarios and implementation ideas with his computational fluid dynamics simulation code, which helped realize major part of my research work. Conversations and discussions with him also broadened my understanding of computational science.

Dr. Eli Tilevich was responsible for the work of adapting Fortran applications using modern software engineering techniques – he convinced me that aspect-oriented mechanisms could be highly useful in implementing modular adaptation of Fortran scientific programs.

Many thanks to the colleagues in Computing Systems Research Lab – Joy Mukherjee, Bharath Ramesh, Ved Vyas Duggirala, Craig Bergstrom, and Hari Pyla – for their help and engaging conversations on systems research. Special thanks to Mike Heffner for his Adaptive Code Collage framework which has been extensively used in this work.

To the members of Soft Lab – Satish Tadepalli, Patrick Butler, Debprakash Patnaik, Yongju Cho, Sheng Guo, Vandana Sreedharan, and Evan Maxwell. They helped make my study lively and enjoyable while I spent two thirds of my Ph.D. time as the only non-data miner in Soft Lab.

To the fellow Korean students in the computer science department for their help in making my days in Blacksburg fun and pleasant. In particular, I am grateful to Dongkwan Kim, Seonho Kim, and Tae-Hyuk Ahn for their support and kindness.

I am fortunate to have the love and support of my family: my wife, Siwon Choi, son, Sehyeon, and daughter, Sebin, have been understanding and patient beyond all reason over the Ph.D. years. My brother, Dae-Ki Kang, and sister, Young-Hee Kang, have been encouraging and caring for so long a time. Finally, I would like to thank my mother for her never-ending support, and my late father for his love and pride in me.

The research in this thesis was supported by the National Science Foundation through award #0615181.

Contents

1 Introduction **1**
1.1 Modular Program Adaptation 1
1.2 Objectives of Implementing Program Adaptation 2
1.3 Research Questions 3
1.4 Outline 4

2 Related Research **6**
2.1 Language and Compiler Support for Adaptation 6
2.2 Middleware Support for Adaptation 7
2.3 Function Call Interception 8
2.3.1 Link-time Wrap 8
2.3.2 LD_PRELOAD 8
2.3.3 Trampolines 9
2.3.4 Advice Weaving in AOP 9
2.4 The Adaptive Code Collage Framework 10
2.5 Dynamic Compilation 12
2.6 Adaptive Algorithm Selection 12
2.7 Computational Steering 13
2.8 Summary 13

3 Dynamic Tuning of Parallel Scientific Codes **14**
3.1 Introduction 14
3.2 Implementing Tuning Support for Existing Parallel Programs 15
3.2.1 Tuning as a Separate Concern 15
3.2.2 Tuning with Collective Consideration of Runtime Factors 16
3.3 Settings for Implementing Dynamic Tuning 16
3.3.1 Target GenIDLEST Parameters for Tuning 17
3.3.2 Input CFD Problems 17
3.3.3 Execution Platforms 17
3.4 Dynamic Tuning Implementation 17
3.4.1 Composition of Dynamically Tuned Software 18
3.4.2 Dynamic Tuning Procedure and Search Strategies 19
3.5 Experimental Results 21
3.5.1 Tuning Cost 25
3.6 Related Work 25

3.6.1 Auto-tuning . . . 25
3.6.2 Computational Steering . . . 26
3.7 Summary . . . 26

4 Reusable Adaptation Patterns Implementation **28**
4.1 Introduction . . . 28
4.2 Adaptation Patterns in Scientific Computing . . . 30
4.2.1 Overview of Adaptivity Schemas . . . 30
4.2.2 Structural Characteristics of Scientific Programs . . . 31
4.3 Adapting Fortran Programs via C++ Aspects . . . 31
4.3.1 Integrating Fortran with AspectC++ . . . 32
4.3.2 Function Call Redirection . . . 32
4.3.3 Generating Fortran to C Wrappers . . . 33
4.4 Implementing Adaptivity Schemas in AspectC++ . . . 34
4.4.1 Obtaining Execution Environment Information . . . 34
4.4.2 Control Systems Schema . . . 35
4.4.3 Algorithm Switching Schema . . . 36
4.4.4 Active Mining of Recommendation Spaces Schema . . . 38
4.5 Evaluation . . . 41
4.5.1 Reusability . . . 41
4.5.2 Software Complexity . . . 43
4.5.3 Performance Overhead . . . 44
4.6 Related Work . . . 44
4.6.1 Multilingual Systems . . . 44
4.6.2 AOP for Scientific Computing . . . 45
4.6.3 AOP for Parallel Programming . . . 45
4.7 Conclusion . . . 46

5 Implementing Runtime Adaptation of Scientific Applications **47**
5.1 Introduction . . . 47
5.2 Adaptive Modeling of Cell Cycles . . . 48
5.2.1 Effective Parameter Estimation . . . 48
5.2.2 Exploration of New Evolution Pathways . . . 49
5.2.3 Effective Simulation through Checkpointing and Rollback . . . 49
5.3 Implementation . . . 49
5.3.1 Adaptation Control . . . 49
5.3.2 GUI for Runtime Simulation Adaptation . . . 50
5.4 Experiments: Dynamic Adaptation of Cell Cycle Simulations . . . 51
5.4.1 Dynamic Parameter Change . . . 52
5.4.2 Simulation Rollback and Continuation . . . 53
5.5 Related Work . . . 53
5.6 Summary . . . 56

6 Conclusions **57**
6.1 Thesis Contributions . . . 57
6.2 Future Work . . . 58

Bibliography **59**

A Program Compilation and Linking **68**
A.1 Stages in Program Build Process . . . 68
A.2 Dynamic Linking . . . 71
A.3 Code Modification Tools . . . 71
A.3.1 System dynamic linker . . . 71
A.3.2 LLL . . . 71
A.3.3 Callee-site modification tools . . . 72
A.3.4 Java tools . . . 72
A.4 Executing ACC Initialization Code before C *main()* . . . 73

List of Figures

2.1 Assembly code patching to intercept *qsort* function calls 11
2.2 Composition of an adaptive application using ACC 11

3.1 Composition of dynamically tuned application with existing code 15
3.2 Hierarchy of data decomposition in GenIDLEST 16
3.3 Problems under consideration for GenIDLEST dynamic tuning 18
3.4 Dynamic tuning of GenIDLEST through the ACC framework 18
3.5 Procedure for dynamic parameter tuning at the intercepted control point . . 20
3.6 Dynamic tuning progress until 3000 time steps for the straight channel problem on Anantham 23
3.7 Dynamic tuning progress until 3000 time steps for the pin fin array problem on System G 24
3.8 Performance comparison of the GenIDLEST code for 10000 time steps 25

4.1 Adaptivity schemas 30
4.2 The ITSOR subroutine and its C wrapper code generated by our wrapper generator 33
4.3 Weaving adaptation aspect code through AspectC++ by exposing Fortran functions in C wrappers 34
4.4 AspectC++ code for obtaining MPI execution environment information . . . 35
4.5 Control systems schema implementation in AspectC++ 36
4.6 Timestep adaptation aspect to improve the stability of GenIDLEST simulations 37
4.7 Algorithm switching schema implementation in AspectC++ 38
4.8 Flow model switching aspect to improve the accuracy of GenIDLEST simulations 39
4.9 Mining of spaces schema implementation in AspectC++ 40
4.10 Dynamic parameter tuning aspect to improve the performance of GenIDLEST simulations 42

5.1 Dynamic adaptation implementation of PET cell cycle simulations using ACC 50
5.2 ODE system that models the cell cycles of frog-egg extracts 51
5.3 Cell cycle simulation of frog-egg extracts 52
5.4 Dynamic parameter change in cell cycle simulations 54
5.5 Runtime simulation rollback and continuation in cell cycle simulations 55

A.1 Compilation and Linking Steps Involved to Create a Process from Source Files 69

List of Tables

1.1 Comparison of program modification techniques 3

3.1 Architectural summary of evaluated platforms for dynamic tuning 18

4.1 ULOC comparison of the GenIDLEST adaptation implementations between the hand-coded and AspectC++ implementations 43

4.2 Complexity comparison of the GenIDLEST adaptation implementations between the hand-coded and AspectC++ implementations using maximum MCC numbers 43

4.3 Execution time (seconds) and overhead measurements of the GenIDLEST adaptation implementations using a pin fin array problem for 500 time steps. 44

Chapter 1

Introduction

Program adaptation is a process of changing the runtime behavior of a program in a different way from its original conception to achieve a certain purpose necessitated by the user. The need for adaptation arises from different application requirements such as performance, stability, and program analysis. Changing system state by modifying global variables and adapting functional behavior of a program module (a chunk of code usually abstracted as a procedure) are typical adaptation operations.

Scientific programming often involves implementing adaptation, whereby a program in execution can change its behavior in response to dynamic changes in the computational properties or the constraints on computing resources. A variety of approaches exist at different levels to implement adaptive behavior. At the algorithmic level, adaptive algorithms [1] or multi-method algorithms [2, 3] can be adopted from the beginning of software design process. At the framework level, adaptation can be supported from tools that perform dynamic algorithm selection or switching to find the best algorithmic option for a given problem at runtime [4, 5]. In particular, support for adaptive execution in a distributed or parallel environment such as a computational Grid has been under active research, which ranges from languages and compilers for specifying adaptation strategies [6–8] to runtime platforms or middleware [9–12].

While these approaches can be useful for developing new adaptive applications, in cases where the behavior of an already existing program needs to be manipulated at a fine-grained level, the required adaption implementation process entails modification over the original program. Hence, implementing adaptation with existing scientific software involves software engineering and maintenance issues with the following important question: "How can application-specific adaptation schemes be effectively implemented and integrated on top of an existing program in a modular way?"

1.1 Modular Program Adaptation

To implement adaptive behavior, original source code can be directly rewritten to include adaptation operations at appropriate control points in the whole program code, and updated as such to produce an adaptive application. However, coping with change is challenging in scientific programming, where old code bases are common and modern programming prac-

tices that encourage modularity and adaptability have not always been used. For instance, numerical software more than a few decades old is not unusual in the listings in the Netlib repository [13]. Even though those codes established stability through numerous bug fixes and performance improvements over their lifetime, it has become inflexible to change their code structure too, unless they had been designed for future restructuring from their inception [14]. The fact that a large portion of scientific codes were written in early versions of Fortran adds to their inflexibility with respect to program changes.

Therefore, even for short-term modification tasks such as adding or changing functional behavior, implementing adaptive decisions on top of existing code can be an onerous task. In simple cases such as replacing a function call with a conditional control structure, the modification replaces the original call by an `if-then-else` statement where either of two functions is chosen depending on the runtime value of a predicate. Yet the rewriting process may become cumbersome in large programs with a complex adaptive plan because the programmer has to locate and update all the places where the modification is needed. Sometimes it requires restructuring of the whole program, which can be quite imposing. Therefore, this issue of adding or changing functionality over existing code in a modular way has been recently identified by the object-oriented programming community as one of the important motivations for Aspect-Oriented Programming (AOP) [15].

However, although object-oriented techniques for modular program development are getting more support in the scientific programming community [16], advanced code insertion features like AOP weaving [17] are still lacking. Binary instrumentation tools can overcome the language dependent issue, but they are not well suited either for extending existing programs with a new program behavior. These tools insert code to existing programs in a compiled binary form, offering clean separation between the original and the new code because the two codes are coalesced at the binary level instead of the programming language level. Nonetheless, since they deal with the native processor instructions at the very lowest level, most of them are developed for advanced program analysis purposes [18–20] such as debugging and profiling rather than as a tool to aid programmers in extending existing programs in a modular way.

Table 1.1 summarizes characteristics of the AOP and binary instrumentation techniques in detail. While AOP frameworks provide an elegant modular way to combine new code with existing programs, all of them are bound to an individually associated object-oriented language. In contrast, while binary instrumentation tools enable language-neutral methods to insert external code into target programs, they are too heavy-weight to be useful for implementing program adaptation in general. Therefore, a novel adaptation approach that gives the best of both worlds is needed.

1.2 Objectives of Implementing Program Adaptation

The goal of the research is to find modular methods to implement diverse adaptive scenarios on top of existing static scientific codes without actually affecting the original program structure. Therefore, the desired method or framework should entail the following characteristics.

- Transparency: The adaptivity code is written and managed as a separate module from the original program. The original code, assumed to have been written in a high level

	AOP frameworks		Binary instrumentation
	C/C++ [21, 22]	**AspectJ** [23]	
MODIFICATION LEVEL	source language	source or binary	machine instructions
LANGUAGE	bound to C/C++	bound to Java	language-neutral
SOURCE AVAILABILITY	always needed	not necessary	not necessary
MODIFICATION METHOD	source-to-source translation	bytecode linking	binary instrumentation
INSERTION TIME	compilation time	compilation or link time	post-compilation time
MAIN USAGE	program behavior extension		program analysis

Table 1.1: Comparison of program modification techniques

language, is not modified. Code insertion should be applied transparently with regard to the original code.

- Language neutrality: The proposed method should not depend on a specific language. In particular, Fortran and C should be supported since the context of the research is scientific programming.
- Usability: While the method involves low-level manipulation of program codes, its interface should be easily accessible to the user, especially in the scientific programming community.

The modularity requirement follows from the AOP world, where insertion of *cross-cutting* code preserves the original class hierarchy. This implies a tool or method that supports the concept of a *module*, which signifies a basic unit of software composition or program control. The language neutrality requirement comes from the world of the binary instrumentation techniques, which work on native machine instructions. This requirement leads to a low-level method, such as systems software or runtime systems support, underneath the programming language layer in the software stack.

The last requirement concerns the usability of the proposed approach. The proposed method should implement a mechanism that provides a high-level view of adaptation strategies while hiding low-level implementation details, so that the users can focus on the design of adaptive scenarios.

1.3 Research Questions

To generalize the proposed method and make it more accessible to the scientific computing community, a high-level model or abstractions to express adaptation strategies are studied in this work. In particular, considering that typical adaptive scenarios in the scientific domain show certain patterns in common, as identified as *adaptivity schemas* [24] for example, the

high-level model should support the patterns to facilitate easy development of adaptive applications. Thus the first question is:

RQ1: How can we design and implement a high-level model to express adaptation schemes in scientific computing, so that it can be easily accessible to scientific programmers?

Algorithms in scientific programs often exhibit different performance characteristics depending on distinct execution environments on which the program runs. As new hardware platforms progress at a rapid rate, auto-tuning or empirical tuning techniques to automatically optimize basic computation algorithms such as matrix-vector operations for a given execution environment are under active research. However, scientific programs are often written using domain-specific algorithms where the computation patterns can not be reduced to such basic linear algebra operations, thus disallowing the use of automatically tuned packages. The following question attempts to examine how program adaptation techniques can be applied for tuning scientific programs an alternative approach.

RQ2: How can program adaptation techniques be utilized to tune algorithmic parameters in parallel scientific programs to match the problem settings and/or the execution environment?

Some adaptation scenarios often consider both precise and loose schemes, where precise schemes has complete adaptation specifications available *before* application launch, while in loose schemes, some part of adaptations are not specified clearly and adaptive decisions are deferred until at runtime, since, the user may have just a vague idea of how the program in consideration evolves and can make adaptation decisions only by monitoring the program's dynamic progress.

Precise schemes alone can be more or less easily implemented because adaptation code can be developed following the scheme and can be combined with existing code to build an adaptive application. However, addition of loose schemes make it more complex to implement because adaptation code for the loose schemes may not be available until after runtime. An approach for tackling this issue will be studied in the context of implementing dynamic model change in cell cycle modeling in the biology domain.

RQ3: How can complex adaptation scenarios with loosely planned schemes be supported?

1.4 Outline

The remainder of this dissertation is organized in such a way that each research question is studied in an individual chapter.

- Chapter 2 presents a broad survey of implementing program adaptation and developing adaptive applications.
- Chapter 3 studies dynamic tuning of algorithmic parameters of parallel scientific applications (**RQ2**).

- Chapter 4 implements high-level abstractions for describing adaptations in scientific computing (**RQ1**).
- Chapter 5 develops an approach to implementing loosely-defined adaptation schemes (**RQ3**).
- Chapter 6 concludes this dissertation and considers possible future directions.

Additional related work specific to individual studies is also presented in the context of each study.

Chapter 2

Related Research

This chapter describes related research in composing adaptive applications and sets the stage for discussion of the proposed research. It covers language and middleware support for adaptivity, function call interception techniques, dynamic binary compilation, and algorithm selection and composition methods.

2.1 Language and Compiler Support for Adaptation

Program Control Language (PCL) [7] is a small language extension that provides mechanisms that one can use to write separate control code that specifies application-specific adaptation strategies at a high level for distributed programs. The expressive power of the language comes from its underlying framework that offers a global representation of the distributed program as a graph, named as the static task graph (STG), of task nodes connected by edges of precedence relationship. A PCL task, defined as a sequence of instructions in a program containing no internal parallelism and no synchronization operations, is the smallest unit for implementing adaptive schemes, and each adaptation primitive of PCL maps to a sequence of graph-changing operations on the STG of the target program. For example, insertion of a new computation in a program is conceptually same as addition of a new task node to the graph. The STG representation also enables remote performance metrics and adaptation strategies to be specified and implemented in global terms by the compiler and runtime system, thus realizing a simple, generalized programming model over a wide range of adaptation strategies in a distributed environment.

Du and Agrawal [8] proposed a Java language extension to help programmers specify *adaptation parameters*, which exposes the degree of flexibility in the quality of the output, for adaptive program execution. By combining runtime information feedback with a static analysis for relating the execution time as a function of the values of the adaptation parameters, a set of optimal parameter values is determined by initial test runs to achieve the best precision while meeting the specified constraints on execution time. The proposed method is rather a static approach for determining optimal parameter values before actual application execution, than a sophisticated dynamic adaptation scheme per se such as runtime algorithm switching.

ADAPT by Voss and Eigemann [6] is an adaptive optimization system that provides a

domain-specific language by which users can describe optimization heuristics to be applied dynamically at runtime. Based on the user-supplied heuristics (e.g., loop unrolling and specifying machine parameters), the ADAPT compiler generates a runtime system that consists of a modified version of the application, which in turn contains two different execution paths for each candidate code section for optimizations. A local optimizer is also generated and launched as a separate thread to communicate with a remote optimizer, detect potential hotspots, and dynamically link in the optimized code variants provided by the remote optimizer.

2.2 Middleware Support for Adaptation

In Grid and cluster computing, there have been extensive effort on runtime platforms for supporting adaptive applications at the level of middleware and network layer to adjust resource management policies or application configuration parameters in response to changes in operating conditions of the environment or based on measured history of application execution time [9–12, 25–28].

The GrADS (Grid Application Development Software) [9] project proposes a *configurable object program* that encapsulates, in addition to the application code, dynamic adaptation strategies to effectively map and schedule Grid resources, for which resource selection and accurate performance models are provided by the GrADS execution framework. CACTUS-G by Allen *et al.* [25] implements dynamic adaptive techniques for efficient execution of astrophysics simulations in distributed, heterogeneous Grid environments. The focus is on automatically adjusting external operating parameters in a distributed execution environment, such as communication message sizes and ghostzone sizes in the grid, with simple heuristic methods on runtime estimates. Condor-G [28] is a mixed technology of Condor [29], a popular job submission and scheduling system for clusters, and the Globus toolkit [30], thereby realizing adaptation of task scheduling and migration in Grid environments. Chang and Karamcheti [26] proposes an adaptation framework that provides a tunability interface component and a virtual execution environment. Based on measured execution time, resource monitoring, and user preferences, the framework permits automatic runtime decisions on when and how to adapt by dynamically choosing a different application configuration. To control the adaptive execution of large stochastic optimization codes, Buaklee *et al.* [10] develops an accurate performance model from a detailed analysis of application execution times with varying configuration parameters, such that an optimal configuration parameters for the distribution of work can be determined without requiring any user-supplied input.

The AppLeS project [11] provides a methodology and software environments for on-the-fly adaptive application scheduling in Grid environments. It has been applied to a variety of domains to result in new applications, each of which consists of domain-specific components and custom scheduling superstructure that is controlled by the AppLeS scheduling agent to monitor available resource performance and generate a dynamic schedule on the target Grid platforms. The applications include adaptive image server selection for SARA (Synthetic Aperture Radar Atlas) applications [31], matrix data decomposition and allocation for Jacobi 2D iterative code [32], and gene sequence partitioning [33].

As their objective is to implement middleware support for adaptation between the ap-

plication and the underlying execution layer, these work are primarily centered around resource management towards efficient utilization of the environment, such as load-balancing and scheduling of application tasks, wherein coarse-grained strategies based on resource constrains or external operating parameters are employed. In contrast, the proposed work aims toward adaptation that can adjust the internal states of a program and its behavior by monitoring the the dynamic progress of the computation.

2.3 Function Call Interception

Function or method call interception (FCI) is a technique of intercepting function calls in order to undertake certain operations before and/or after the function executes, or even to entirely replace the call. FCI is typically used for debugging purposes and performance analysis such as tracing and profiling. It is also used to change the program behavior by modifying the parameters or return value of an intercepted function. In the AOP frameworks, for example, FCI is used to implement advice weaving on join points, such that a piece of associated aspect code is executed whenever the join point is reached by the program execution.

FCI can be implemented at different levels of the runtime software stack and simple implementations include link-time wrap and LD_PRELOAD [34]. Although it is hard to cover the whole set of FCI techniques, we describe a set of basic programming primitives and constructs that are found useful to implement FCI, such as trampolines and aspect weaving.

2.3.1 Link-time Wrap

Link-time wrapping, a functionality offered by the system (static) linker on Unix systems, is to wrap a function by renaming its *symbol name* so that it is not directly callable from other program components. At *program link time*, the linker replaces the wrapped function's symbol name in other components with something else, which becomes the wrapper name, and the programmer is supposed to provide the body of the wrapper in which the original function can be called. For example, the GNU linker, when passed the option '–wrap foo' to wrap the function `foo`, substitutes `__wrap_foo` for the `foo` references to generate the output. The linker also creates the `__real_foo` symbol for the actual `foo` function, which can be used by the programmer to call `foo`.

2.3.2 LD_PRELOAD

LD_PRELOAD is an environment variable that specifies dynamic libraries to be loaded into an application's address space early at load time, so that the system dynamic linker looks up the definitions for unresolved symbols (e.g., externally declared functions) in LD_PRELOADed libraries before in non-PRELOADed ones. Therefore, LD_PRELOADing can be used to redirect function calls, such that a different implementation (but with the same signature) in a LD_PRELOADed library substitutes for the original function. Furthermore, inside the redirected call, the programmer can make a call to the original function through the dynamic

linking and loading APIs (i.e., `dlopen()` and `dlsym()`), which can realize function wrapping.

LD_PRELOADing works on binary shared libraries (commonly referred to as dynamic shared objects (DSO)), which have the usual advantages of binary components such as language-independence, intellectual property protection, and flexible load-time configurability [35]. However, since LD_PRELOADing requires the code to be in a shared library form, it can be inconvenient for standalone applications because the programmer needs to separate out the relevant part of the code to apply LD_PRELOADing, recompile it as position-independent [36], and generate the binary in the shared object form. Also, LD_PRELOADing can be sometimes unreliable [34].

Details of linking (static or dynamic) and shared libraries are described in Appendix A.

2.3.3 Trampolines

In binary rewriting FCI techniques, a *trampoline* is a piece of code which consists of a few instructions replaced from the original target function and a jump instruction back to the remainder of the target. The empty place in the target where instructions were removed opens the door to realizing FCI such that a jump to an intercepting function can be inserted instead and the execution control can be transferred. Then the intercepting function can perform a user-specified operations before executing the target. At the exit of the interceptor, it may or may not jump to the trampoline depending on whether to execute the target.

Trampoline techniques (also called *probe-based* techniques) are easier to implement compared to JIT translator-based methods because all it does is to replace the beginning of a target function with a branch to an interceptor and store the removed instructions in a trampoline, while JIT-based techniques need to have a compiler along with a VM for executing the translated code.

The use of trampolines adds performance overhead due to the increased level of indirection from multiple branches. Besides, they cannot be used if a target is not big enough to hold branching instructions to an interceptor. For instance, Detours library [37] does not work if the target function is fewer than 5 bytes which is required to insert an unconditional jump instruction. Also, the transparency is not preserved since the original instructions are overwritten in memory.

2.3.4 Advice Weaving in AOP

Aspect-Oriented programming (AOP) [15] is a programming paradigm in software engineering which attempts to help programmers in the software design process through the principle of *separation of concerns*. There are aspects of software programs, such as logging, security, and performance, that cannot be easily captured by typical inheritance hierarchy of object-oriented programming techniques. AOP helps to modularize those concerns spread across the whole program into *cross-cutting concerns*. A major way of implementing a cross-cutting concern in AOP is to use *advice*, which is a piece of code written by the programmer to be performed at those program execution points specified as a *pointcut*. When a running program reaches a *join point* specified in a pointcut and if the point is at a method call, the call is intercepted and the program control is transferred to an associated advice code such that it executes. AOP languages offer language constructs to express a pointcut and advice code,

which can be called before or after executing a target method. Also an advice can entirely replace the original target call.

AOP languages and frameworks implement FCI through *advice weaving*. During the advice weaving process, AspectC [21] and AspectC++ [22] perform source-to-source translation on an aspect code, and generate standard C and C++ code respectively. The translated C/C++ code is fed to a normal C/C++ compiler along with a target application code to form an FCI-enabled executable at last.

AspectJ [23] provides a weaving compiler that accepts both the source and bytecode files to create an advised bytecode. The front-end of the compiler is an extension of Java compiler, which can compile AspectJ code into standard Java class files where each advice declaration is compiled into a normal Java method, appropriately annotated with extra Java bytecode attributes [38] to store information of the corresponding advice. Although the *ajc* compiler in [17] can only perform static instrumentation, newer versions [23] are able to perform dynamic weaving too.

Static advice weaving in AspectJ consists of three individual steps: identification of *join point shadows*, matching *point cut designators (PCD)* against the shadows, and aspect code insertion. In AspectJ, a *join point* is a point in a dynamic call graph of program execution for possible advice code insertion. Every dynamic *join point* has a corresponding static place in the source code. The first step of weaving is to identify every potential shadow in the bytecode. For this purpose, Hilsdale [17] defines 11 kinds of shadows which include method-call/execution and field-get/set for example.

Each joint point shadow in each class found in the first step is matched against each *PCD* specified in the aspect code. If a PCD matches a shadow, the advice code associated with the PCD is inserted to the shadow. However, the matching process might not be complete since the PCDs may depend on the dynamic state at the *join point*. In order to handle this mismatch, a dynamic test should be added that performs the dynamic part of the matching.

The last step is to actually insert the advice code at the matched shadow. Sometimes the program context at the shadow should be exposed and preserved appropriately before inserting the code, because the advice or the advised code may need the context information for its correct operation. The way an advice is inserted and how the bytecode is transformed accordingly depends on the type of an advice. AspectJ defines five different advice types, each of which has its own implementation of weaving.

2.4 The Adaptive Code Collage Framework

Adaptive Code Collage (ACC) [39, 40], formerly called *Invoke*, is a framework for easy function call interception and manipulation, developed to realize a runtime framework for adaptive compositional modeling. As shown in Figure 2.1, where the calls to the *qsort* function are intercepted, ACC implements FCI at the assembly language level by replacing every *call* instruction in a module to be instrumented with a call to its own interception handler, which in turn accepts the target function address as its argument.

The capabilities of the ACC framework are,

- Function interception : User-specified function calls can be intercepted before or after

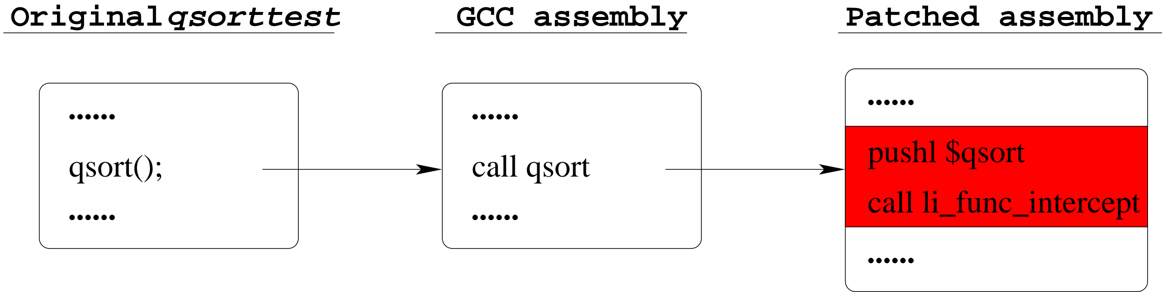


Figure 2.1: Assembly code patching to intercept *qsort* function calls

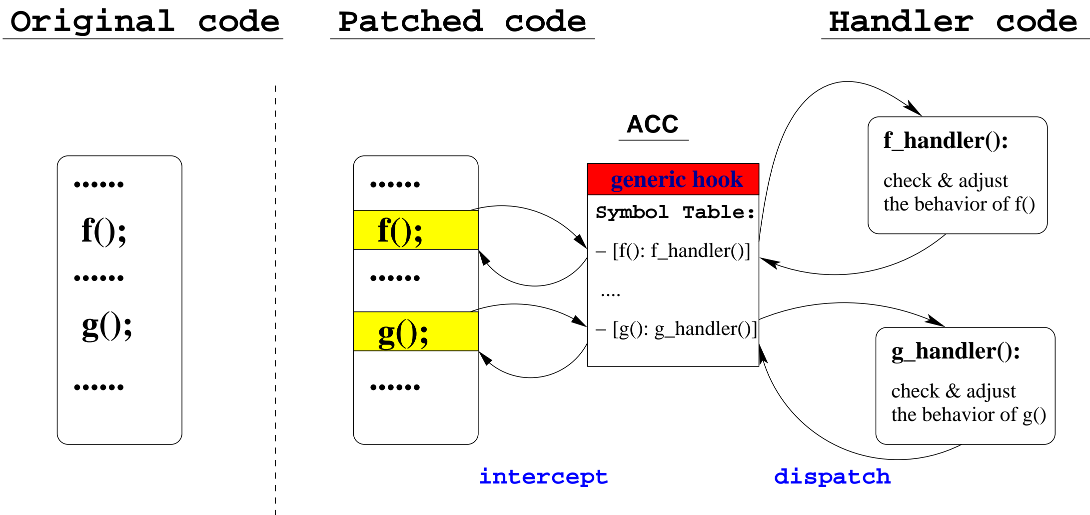


Figure 2.2: Composition of an adaptive application using ACC

the actual call through assembly code patching. This means the basic unit to support code insertion is a procedure.

- Registered callbacks : A piece of code can be registered with a function such that the code can be performed whenever the associated function is intercepted.
- Parameter manipulation : Function call parameters can be accessed and modified through stack manipulation before the call is actually performed, which allows another degree of flexibility in modifying the original program behavior.
- Function remapping : Through sophisticated stack manipulation, the entire parameter list of a function can be remapped to a new function with different signature.

By specifying a target function to be manipulated by ACC, the programmer essentially defines an adaptive control point over the original program, where newly developed modules can be introduced to maneuver the program toward the intended adaptive behavior. Thus, as Figure 2.2 shows, composition through ACC enables one to separately reason about application-specific adaptive strategies, factor them out in a centralized code, and plug in the adaptation code at control points to build an adaptive application.

2.5 Dynamic Compilation

Dynamic binary instrumentation tools [19, 20, 41] perform just-in-time (JIT) compilation on instructions being fetched from a running process in memory in order to insert analysis code into them during the compilation process. The instrumented instructions are then executed by a virtual machine (VM) in the tool. Since the level on which the tools operate is low-level machine instructions, very fine-grained program control is possible and the tools are more suitable for debugging and analyzing program behavior at the instruction level such as tracing or profiling.

The most attractive characteristic of the *jit-based* tools is that the dynamic compilation technique they use is transparent. Unlike the trampoline-based approach where application instructions are overwritten with trampoline code, which in turn may affect original addresses or contents of memory and registers, dynamic compilation preserves the original program state because it translates instead of overwriting the code. The user running the instrumented code sees the same addresses and contents as in running an uninstrumented code.

One major drawback of dynamic compilation is performance overhead. Since the compilation and instrumentation is performed when the application actually runs, it directly affects the application's runtime and slows down the program execution, which may grow by an order of magnitude. Hence optimizing the extra performance overhead is inevitable. For example, Pin's optimization techniques include inlining, register liveness analysis, and instruction scheduling [19].

2.6 Adaptive Algorithm Selection

Most scientific computations are modeled as nonlinear simulations, which depend to a large extent upon the efficiency of linear system solvers. However, it is impossible to uniformly rank a diverse set of linear solvers in terms of cost efficiency and convergence behavior within broad range of large, sparse, nonsingular systems, because the numerical characteristics of the linear systems can change as nonlinear iterations progress. This situation has motivated researchers to explore using multi-method solvers as a way to improve the performance [2, 42].

Bhowmick *et al.* [3, 43] has proposed two different ways to implement multi-method linear solvers. The *composite solvers* apply a sequence of preconditioned base methods to a given linear system until the desired convergence is achieved. The optimal ordering of applying base methods is constructed such that the *utility ratio*, the ratio between the completion time and the reliability rate, of each method is in increasing order. The *adaptive solvers* select the solution method dynamically to match the changes in numerical properties of the problem. The automated selection uses metrics such as time per iteration and convergence rates in both linear and nonlinear iterations. The adaptive solver scheme assumes the base methods are ranked in terms of cost and quality for a given class of problems, such that the scheme begins with a lowest ranked method that leads to a lower-accuracy result in a shorter amount of time, and switches to a more powerful method when the linear system becomes more difficult to solve.

Thomas *et al.* [44] implements an algorithm selection framework for use in STAPL [4], a

C++ library for developing parallel programs. It uses machine learning to analyze benchmark data collected during installation, and selects best algorithmic option for a problem instance at runtime. Johnson and Eigenmann [45] utilizes AI planning techniques in a compiler to automatically compose a sequence of context-sensitive operations from a programmer-defined abstract algorithm. In contrast, Yu *et al.* [5] proposes an adaptive framework, where parallel reduction algorithms are replaced dynamically to match the characteristic changes in a running program.

2.7 Computational Steering

Computational steering is the practice of manually altering parameters of scientific simulations at runtime through interactive feedback from the user [46–48]. Since computational steering enables dynamic parameter change decisions to take effect at runtime, it can be beneficial for runtime performance tuning and "what-if" studies for certain problems without requiring stop-and-restart of computation from scratch. However, sophisticated data visualization is required to help guide the user in analyzing the progress of a running simulation, which often takes significant programming effort to implement. Furthermore, implementing efficient middleware or runtime support to instantly update the program states at the computation backend with the user's feedback at the steering fronted is a major challenge in large, distributed systems such as the Grid – the main execution environment of computational steering applications – where communication between remote sites becomes a main source of performance bottleneck [49].

2.8 Summary

FCI techniques are most commonly used to change the behavior of existing programs without directly modifying the original source code. Trampolines provide a language-neutral way to intercept function calls in native executables. However, they are unreliable because they might not work for small-sized target functions. Aspect weaving in AOP is a very sophisticated method for inserting new codes into an already established class hierarchy, but each AOP framework is bound to its associated language. While dynamic compilation techniques overcome the language neutrality limitation, they are too fine-grained to be useful for the purpose of implementing adaptive applications with existing codes. Adaptive algorithm selection methods are attractive in the scientific programming domain. However, the approach is not modular because altering the original code is unavoidable.

The proposed method in this research features both modularity and language neutrality in composing adaptive applications with existing codes, achieving the best of both worlds of AOP and binary rewriting methods. Moreover, the research work covers practical implementations of a set of characteristic adaptive scenarios surpassing just dynamic algorithm selection.

Chapter 3

Dynamic Tuning of Parallel Scientific Codes

3.1 Introduction

Algorithmic parameters can have a critical influence on the performance of a scientific application. A typical example involves domain decomposition methods, where a large problem domain is partitioned into small subdomains or blocks, with a dominant step in the algorithm corresponding to independent solves on each of the subproblems, which are then combined in some fashion to drive the global solution forward. Since a carefully chosen subdomain size (e.g., chosen to match the execution platform's memory hierarchy in some way) can lead to a significant speedup, tuning of the decomposition method is a common optimization strategy in this setting. Manual tuning of algorithm parameters not only is time-consuming in scientific computing, where several days or weeks of simulation is not unusual, but also requires a substantial level of understanding of both the target algorithm and the underlying hardware. This is particularly difficult with today's fast evolving hardware architectures such as multi- or many-core CPUs with complex memory hierarchies. Therefore, tuning support to automatically optimize scientific codes over parameter search spaces is becoming more relevant [50, 51]. For instance, vector and matrix operations – basic computational kernels in numerical computing – are a prime target for automatic performance tuning (or 'auto-tuning'). These methods generate highly optimized codes by parameterizing performance-critical function and benchmarking the target system, thereby automatically determining the best possible configuration [52, 53].

Not all applications lend themselves to traditional approaches to auto-tuning, however. Scientific programs are often written using domain-specific data structures and algorithms, where patterns of computation cannot be mapped to well-studied basic linear algebra operations, thereby preventing direct application of automatically optimized software libraries. In particular, tuning support for algorithmic parameters in existing scientific codes is lacking, where even a modest effort in parameter tuning can provide significant speedups. Another limitation with conventional auto-tuning approaches is that they mainly consider only static hardware characteristics of a single machine and do not include runtime factors that develop when an application executes with an actual input problem on a given computing platform.

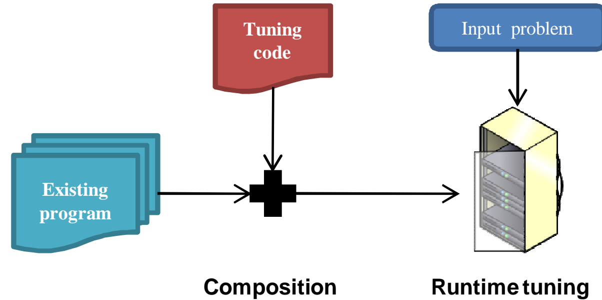


Figure 3.1: Composition of dynamically tuned application with existing code

In addition, in the high-performance computing context, parallel runtime factors arising from communication patterns and synchronization costs, which are a critical determinant of application performance, are not usually considered by traditional auto-tuning approaches.

To address these issues, this work develops a dynamic method for tuning algorithmic parameters of existing scientific codes. By using the ACC framework that supports language-independent composition (Section 2.4), the developed method supports separate development and transparent integration of tuning code for a given program, so that at runtime the newly inserted code can perform application-specific tuning operations, such as dynamic search of parameter space and performance behavior analysis, to determine optimal parameter values for a given input. Specifically, we target existing programs whose performance-critical algorithms show distinct performance behavior depending on runtime factors and, at the same time, are not well supported by auto-tuning techniques, although the method can be applied to any program in general with tunable algorithmic parameters.

3.2 Implementing Tuning Support for Existing Parallel Programs

To implement tuning support for existing codes, we essentially take a compositional approach to enhance a given program with an application-specific plug-in module that performs dynamic tuning on the program's algorithmic parameters. The key concepts are described here.

3.2.1 Tuning as a Separate Concern

While current optimization or tuning techniques are certainly beneficial for basic numerical operations (e.g., matrix-vector multiplication), they are not directly applicable to programs with application-specific algorithms that do not use such operations. Without support for tuning, the user of such applications often employs heuristics or parameter values that are empirically proven "sufficient", which, however, may turn out to be sub-optimal for different execution platforms. On the other hand, manually implementing application-specific tuning strategies for an existing program requires rewriting relevant parts of original program code base, where the modification process can disturb the program's original structure and design

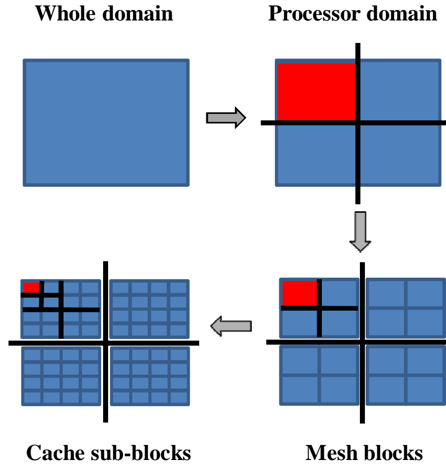


Figure 3.2: Hierarchy of data decomposition in GenIDLEST

purposes by intermixing both new and old codes and blurring logical boundaries between components.

To secure modularity in the design and implementation of tuning strategies over existing programs, we treat tuning as a *separate concern* in evolving software, where tuning plans are organized separately and their implementations are written and managed in independent modules while the main program maintains its original form and structure, unaffected by individual tuning efforts. Tuning code modules are then later plugged into the main program through a compositional framework, as illustrated in Figure 3.1, which combines both codes to generate an application with a newly added tuning capability that optimizes parameter values in computation algorithms of interest at runtime.

3.2.2 Tuning with Collective Consideration of Runtime Factors

To account for runtime factors that affect performance behavior depending on different properties (e.g., size) of actual problem instances and parallel characteristics of a given execution platform, we use a dynamic method for tuning target programs. Specifically, we implement a dynamic tuning procedure that searches for optimal values of application-specific algorithmic parameters in the beginning of program execution by periodically measuring and analyzing the runtime performance profile.

3.3 Settings for Implementing Dynamic Tuning

We showcase the dynamic tuning method in the context of GenIDLEST [54], a CFD simulation code written in Fortran 90 with MPI to solve the time-dependent incompressible Navier-Stokes and energy equations. Specifically, we aim to tune the multilevel Additive Schwarz preconditioned Krylov method used to solve the elliptic pressure Poisson equation which accounts for a large fraction of the total computational time. For flexibility, the preconditioner provides many options and parameters which can be tuned for optimal performance. The target parameters for dynamic tuning are described in the following.

3.3.1 Target GenIDLEST Parameters for Tuning

Cache sub-block size in Schwarz preconditioner

Figure 3.2 shows the hierarchy of the data structure in GenIDLEST in which the computational domain is decomposed into overlapping computational blocks. Each computational block is further divided into smaller sub-blocks. The Schwarz preconditioning is applied by iteratively smoothing the solution in each of the sub-blocks using a multilevel approach. The size of the sub-blocks directly impacts "cache performance" which is critical to the overall time-to-solution. However, because of the complexity of hierarchical memory subsystems and the mathematical characteristics of the system matrix, optimal sub-block sizes can vary substantially on different chip architectures and for different physical problems. In the GenIDLEST code, the input parameter set (ni_blk,nj_blk,nk_blk) represents the number of sub-blocks (and hence the size of each sub-block) in the x-, y-, and z-directions, respectively.

Inner relaxation sweeps

The inner relaxation sweep input parameter, nswp_in_blk, specifies the number of sweeps or iterations performed by the smoother each time a sub-block is visited. In general, increasing this value improves cache performance. However, taking more sweeps also translates to more floating point operations which may not have the desired favorable effect on convergence characteristics and hence could increase the CPU time. The default value is set to 5 in complex flows. In simpler flows, higher values may give better overall CPU time.

3.3.2 Input CFD Problems

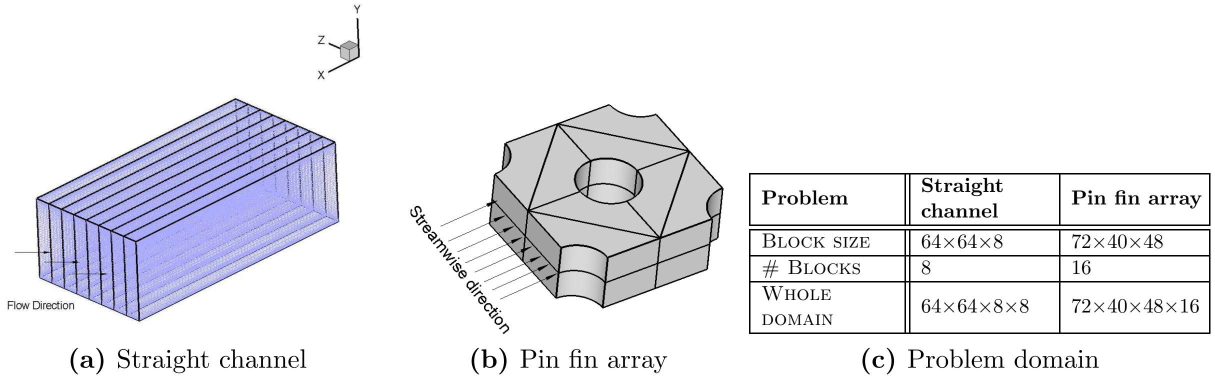
Two CFD problems are considered that show distinct physical characteristics to evaluate our tuning method: a turbulent straight channel (Figure 3.3(a)) and a pin fin array (Figure 3.3(b)). A detailed understanding of flow and heat transfer characteristics of these problems are important in various applications and affect the design and use of these applications. The computational domain of each problem is shown in (Figure 3.3(c)). For the turbulent straight channel problem as an example, grids of $64 \times 64 \times 64$ computational cells were used in the x-, y-, and z-directions, respectively, which are divided in the z-direction into eight $64 \times 64 \times 8$ blocks.

3.3.3 Execution Platforms

Two cluster systems are used to perform dynamic tuning of GenIDLEST simulations, called Anantham and System G, respectively. A summary of architectural features of each of the evaluated systems is shown in Table 3.1.

3.4 Dynamic Tuning Implementation

The development process of the developed dynamic tuning method involves identifying control points at which an existing program will be instrumented to plug in tuning code, designing and implementing dynamic search strategies in a separate module, and combining



Problem	Straight channel	Pin fin array
Block size	64×64×8	72×40×48
# Blocks	8	16
Whole domain	64×64×8×8	72×40×48×16

(a) Straight channel **(b)** Pin fin array **(c)** Problem domain

Figure 3.3: Problems under consideration for GenIDLEST dynamic tuning

Cluster	Anantham	System G
CPU	AMD Opteron 240	Intel Xeon E5462
Clock (GHz)	1.40	2.80
# Sockets	2	2
# Cores per Socket	1	4
L1 Data Cache	64KB	32KB
L2 Cache	2×1MB	4×6MB (shared by 2)
Memory	1GB	8GB
Interconnect	100Mbps Ethernet	40Gbps InfiniBand
MPI&Compiler	MPICH2 1.0.8 with GNU Compilers 4.2.5	OpenMPI 1.2.8 with Intel Compilers 11.0

Table 3.1: Architectural summary of evaluated platforms for dynamic tuning

the implemented tuning code into the original program. This section describes these development aspects of our dynamic tuning method.

3.4.1 Composition of Dynamically Tuned Software

To allow separate development of application-specific tuning code and to compose a dynamically tuned application with a given program, *tuning control points* are identified in the original code, such that the execution control is intercepted and transferred to the tuning module.

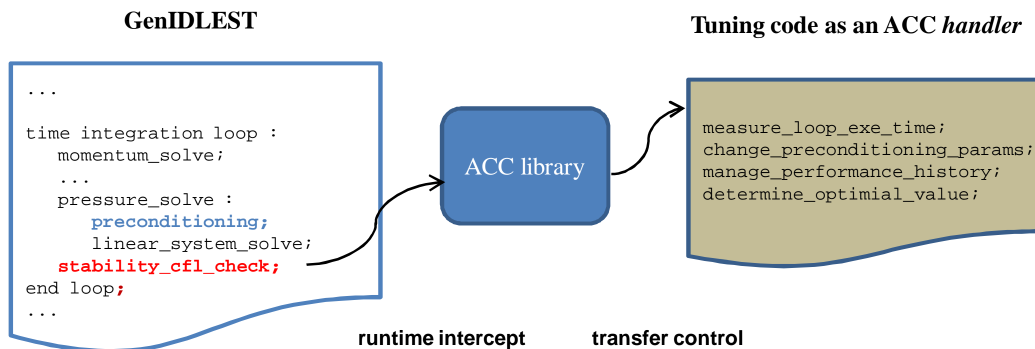


Figure 3.4: Dynamic tuning of GenIDLEST through the ACC framework

These control points are the places at which application performance is regularly measured and the considered algorithmic parameters are updated by tuning operations. While the programmer is responsible for determining control points in a program, they can be conveniently chosen at the end of the simulation loop in typical scientific applications, such that tuning operations can be performed at regular intervals. However, for concurrently executing processes in a parallel environment, tuning global parameters requires global synchronization, which can degrade application performance. To mitigate the potential performance slowdown, we use a piggybacking technique [55] on existing global functions that execute synchronously near the loop end in the original program, thus effectively amortizing the synchronization overhead by placing separate barriers close together. For the GenIDLEST code, we choose as the tuning control point the stability check function, `calc_cfl`, which is called at the end of the time integration loop, and which calculates CFL (Courant-Friedrich-Levi) numbers at every preset number of iterations.

To plug in tuning code to GenIDLEST, the ACC framework is used to take advantage of its language-independent composition capability that operates on the compiler-generated assembly source code. Therefore it does not require direct modification of existing code. Therefore, we define an ACC *handler* function for `calc_cfl` and register it through the ACC registration API, so that the intercepted calls to `calc_cfl` are diverted to the associated handler, which then performs separately implemented tuning operations. Figure 3.4 shows the tuning control point in GenIDLEST; the execution control is intercepted at runtime by catching the `calc_cfl` calls and passed to the tuning module through the ACC library.

3.4.2 Dynamic Tuning Procedure and Search Strategies

A simple dynamic tuning strategy is to use a two point measure-and-compare scheme. At the first measurement point, execution time for a certain computational interval is measured with the current value of an algorithmic parameter. A new value of that parameter is then selected and the same interval is timed again at the second measurement point, so that the performance between the two points can be compared, and so that the relative improvement or degradation in performance, as a function of the algorithmic parameter, can be estimated. However, this two point scheme may easily fail to determine the right path to a potential optimum in the parameter search space, since, particularly in scientific simulations, physics of the simulated problem (e.g., turbulence in CFD simulations) is not known *a priori* and can change at different phases of computation, resulting in changes in performance behavior even for the same parameter value. To complement the two point tuning scheme to cope with unknown computational progress, we use one more measurement point to obtain the slope of change in execution time for the current parameter value. This slope information is used in the search heuristic, in a way similar to the threshold function in empirical optimization [56].

Figure 3.5 depicts the developed dynamic tuning procedure, which comprises four stages (three measurement stages and one compare-and-decide stage). For each algorithmic parameter under tuning consideration, exploring a point in the search space takes four stages: stage 1 (line 4–7) initiates measuring performance with the current value, stage 2 (line 8–11) completes measurement and repeats one more examination to measure the slope in performance behavior for the same parameter value, stage 3 (line 12–18) completes the second measurement and starts measurement with a newly selected parameter value, and stage 4

```
1  Parameters p_1, p_2, ..., p_i, ..., p_k
2  if p_1, ..., p_{i-1} are marked tuned and p_i is not then
3      switch stage do
4          case 1
5              elapsed_time ← 0;
6              start measuring elapsed time with the current value of p_i;
7              break;
8          case 2
9              elapsed1 ← elapsed_time, elapsed_time ← 0;
10             start measuring elapsed time with the current value of p_i;
11             break;
12         case 3
13             elapsed2 ← elapsed_time, elapsed_time ← 0;
14             choose a new value for p_i in the search space;
15             broadcast the new value of p_i from root to all processes;
16             change p_i with the received value;
17             start measuring elapsed time with the new value of p_i;
18             break;
19         case 4
20             elapsed_new ← elapsed_time;
21             compare performance of the new and the old values of p_i;
22             determine either the old or the new value as the current optimum;
23             broadcast the selected optimum of p_i from root to all processes;
24             change p_i with the received value;
25             update the search space of p_i;
26             if the search space of p_i is empty then mark p_i as tuned;
27             break;
28         otherwise break;
29     end
30 end
```

Figure 3.5: Procedure for dynamic parameter tuning at the intercepted control point

(line 19–27) completes the new measurement and compares performance behavior based on the measured history and decides whether the search point is better than the current one.

For stages greater than 4 (line 28), the tuning procedure performs no operations. These empty stages are used to control how fast (or slow) each tuning round for one search point should perform. With more stages than 4, for instance 10, the tuning procedure just breaks and returns to the main simulation code for stages 5 to 10, essentially slowing down the whole tuning process.

Tuning the Cache Sub-block Size Parameter

An important issue in dynamic tuning is balancing the trade-off between tuning cost and program performance: spending too much time on exhaustive search of parameter space for absolutely optimal tuning points can adversely affect the overall time-to-solution of the simulation with diminishing returns, whereas searching only a small part of parameter space to reduce tuning cost can end up with sub-optimal results, failing to achieve the desired speedup. Since the search space of the sub-block parameter is 3-dimensional, and can grow

quite large with only a small increase in problem size, exhaustive search of every combination of (x,y,z) values is not feasible. To reduce the search space, we first eliminate considering the z-direction (nk_blk) of the parameter space and use the value as specified by the user, since, once ni_blk and nj_blk are tuned, further variations in the sub-block size by changing the z component will be small and are unlikely to cause any drastic changes in the GenIDLEST performance behavior.

Secondly, to further reduce the search space of the remaining (ni_blk,nj_blk) pair, we employ a two-phase tuning scheme that performs *coarse search* in the first phase and *fine search* in the second phase. In the coarse search phase, the 2-dimensional pair is treated like a 1-dimensional variable by changing both ni_blk and nj_blk together by same factor (such as divide-by-2). In addition, exploiting the fact that after a certain optimal point the GenIDLEST performance becomes worse as the number of sub-blocks increases (the sub-block size becomes smaller), we decrease the number of sub-blocks (so that the sub-block size becomes bigger) in the coarse search phase, starting from large initial values for the pair, which will exhibit a certain period of improved performance throughout the search. Once a point is reached where performance starts deteriorating, the search process backs off to the previous point and continues by changing both ni_blk and nj_blk with a decrement of 2 at each exploration step, in an attempt to gradually converge to an optimum in the subspace of (ni_blk,nj_blk). With the (ni_blk,nj_blk) value found in the coarse search phase, the fine search phase starts to tune nj_blk further, with ni_blk's value fixed now, by searching a 4-point neighborhood in the y-direction with an increment (or decrement) of 2, and selecting the point in the neighborhood that shows the best performance.

Tuning the Inner Relaxation Sweep Parameter

Once tuning of the cache sub-block parameter is complete, we start tuning the inner relaxation sweep parameter. Unlike the search space of the sub-block size parameter that is directly dependent on the input problem size, the possible number of smoothing iterations on each sub-block is not limited or restricted by the problem size, which allows flexibility in defining its search space. In such a case, practical experience with the target code, together with understanding of tuned algorithms, helps to narrow down the parameter search space, enabling effective search strategy design. Therefore, we assume the default value of 5 as a reasonably obtained one from GenIDLEST simulation practice, and use it as a starting guess. In addition, we examine the four-point neighborhood of the default value by defining the search space as {1,5,10,15,20}.

3.5 Experimental Results

Figure 3.6 shows the dynamic tuning progress of GenIDLEST for the straight channel problem on Anantham for the first 3000 time steps. In Figure 3.6(a), the thick gray solid line represents the elapsed time measured at every 50 steps during the tuning process. Elapsed time plots measured for a set of different parameter configurations without tuning are shown together for comparison. Specifically, *blk2* represents a simulation configured with the cache sub-block parameter set to (2,2,1) and the inner relaxation sweep parameter to 5, which

can be designated as a combination {(2,2,1),5}. Similarly, *blk4* represents a simulation with {(4,4,1),5}, *blk8* for {(8,8,1),5}, *blk16* for {(16,16,1),5}, and *blk32* for {(32,32,1),5}. The parenthesized number following each *blk* is the corresponding cache sub-block size for the configuration.

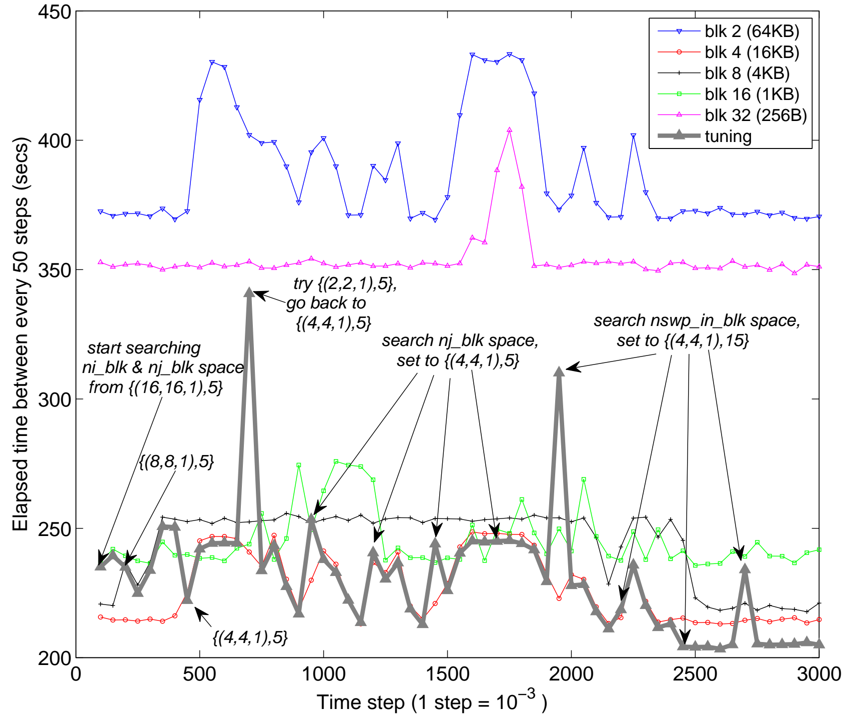
In Figure 3.6(a), the dynamically tuned GenIDLEST simulation progresses as follows. With an initial value of {(16,16,1),5} for {(ni_blk,nj_blk,nk_blk),nswp_in_blk}, the enhanced GenIDLEST program starts tuning first with the sub-block parameter by periodically measuring elapsed times and exploring a new parameter value, where the parameter is changed first to (8,8,1), then to (4,4,1), and so forth at each exploration step according to the search scheme. The sub-block parameter finally settles down to (4,4,1) at time step 1700. Compared with other plots for fixed sub-block parameter values in the figure, the tuning progress plot closely follows the execution time profile with the same parameter value after each update of the parameter, which shows the effectiveness of our dynamic tuning method. Tuning of inner relaxation sweep parameter follows after the sub-block parameter is handled, which completes with the value 15 at time step 2700.

Figure 3.6(b) shows cumulative elapsed time (i.e., consumed execution time) plots for the tuned and the non-tuned simulations. It shows that the performance of the tuned simulation is overall better than all the non-tuned simulations except for the *blk4* configuration, which performs slightly better.

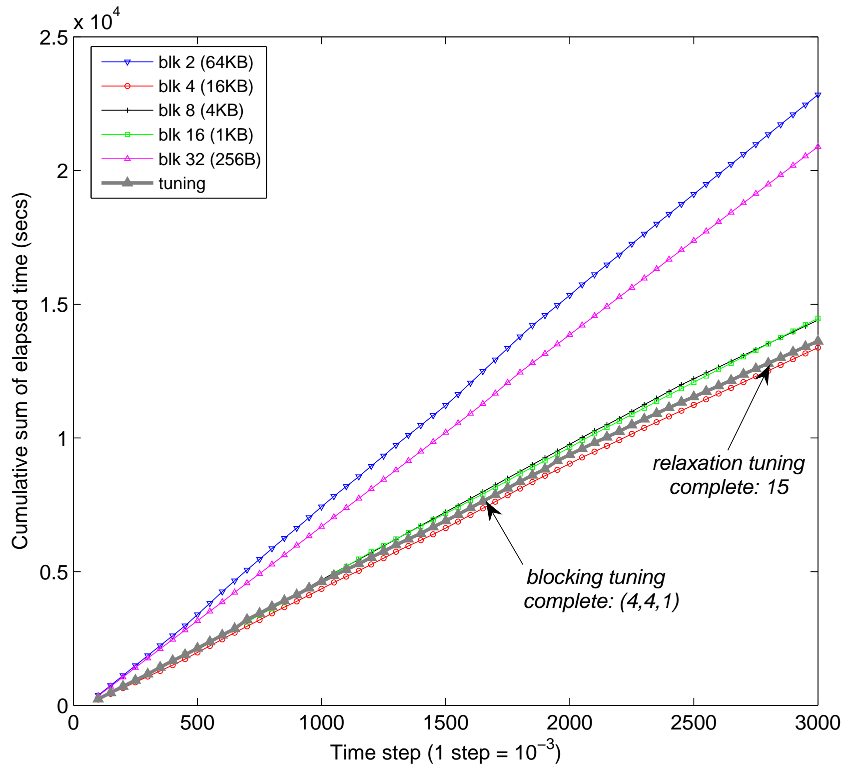
Similarly, Figure 3.7 shows the GenIDLEST tuning progress for the pin fin array problem on System G. In Figure 3.7(a), the tuned GenIDLEST simulation starts with an initial value of {(16,16,8),5} for the cache sub-block parameter and the inner relaxation sweep parameter. The tuning code module periodically measures the loop performance while it explores the parameter space, thus continuously choosing a better parametric option for the ongoing computation. Finally, at time step 2950, the tuning process completes with the parameter values {(2,4,8),5}. Figure 3.7(b) shows cumulative elapsed time plots for the tuned and the non-tuned simulations for the pin fin array problem on System G. Similar to the straight channel problem on Anantham case, the dynamically tuned simulation performs better than all the non-tuned simulations except for only one case, the *blk2* configuration.

To evaluate our method, we performed dynamically tuned GenIDLEST simulations with each of the CFD problems for 10000 time steps and measured their execution times on each cluster. For comparison, we also measured the performance of GenIDLEST simulations with fixed parameter sets (without tuning). The fixed-parameter simulations were performed for each of the four configurations from *blk2* to *blk16*. As well as the total execution time for 10000 steps, elapsed times between every 50 steps were also measured for each simulation. Figure 3.8 compares the performance of dynamically tuned GenIDLEST simulations against that of the simulations without tuning. In the figure, 'tuning' represents the total execution time of the tuned simulations, 'best' represents a hypothetically best time that can be achieved by switching between the four configurations (from *blk2* to *blk16*) at every 50 steps. The 'best' time is calculated by taking the minimum elapsed time among the four configurations at each 50th step and then by summing up these piecewise minimum times for the whole 10000 steps. Finally, the 'expected' time represents an expectation value for the total execution time by randomly choosing a fixed parameter set out of the four configurations.

For the straight channel problem, our method performed better than 'expected' by 26% and slightly better than 'best' by 2% on Anantham, and also outperformed both 'expected'

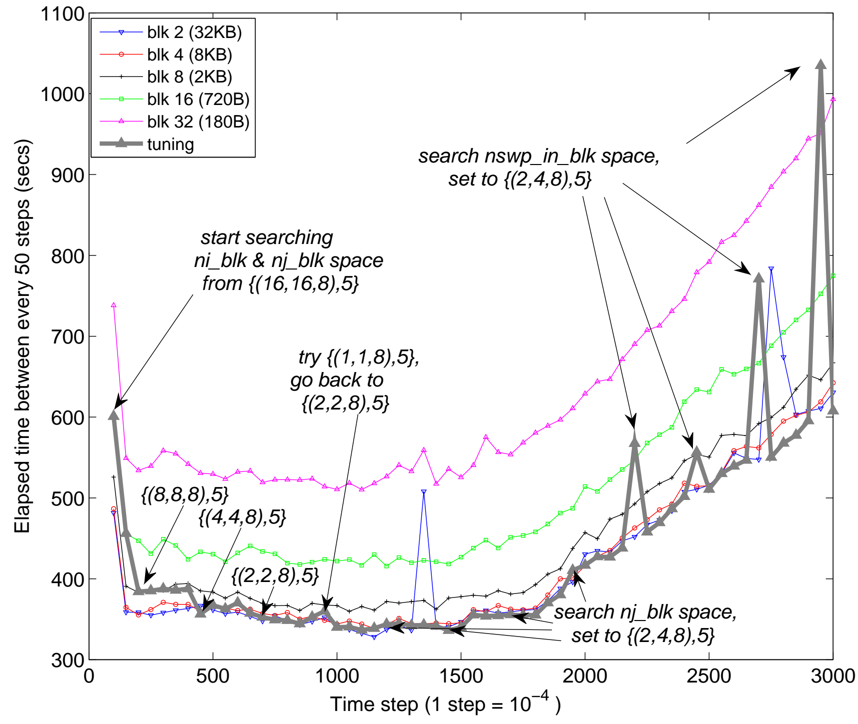


(a) Elapsed time between every 50 steps

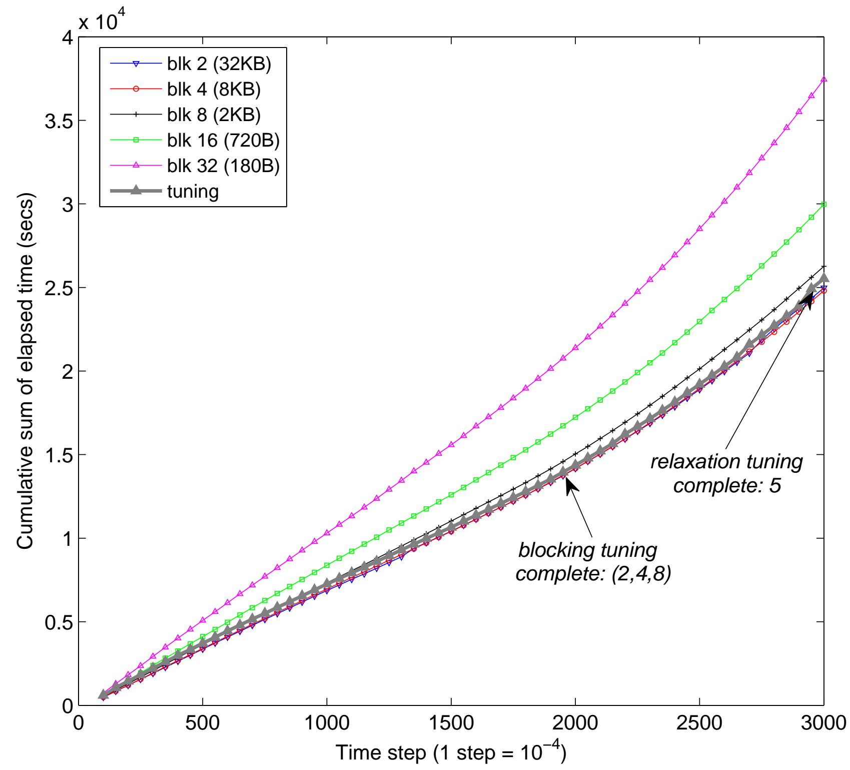


(b) Cumulative elapsed time at every 50 steps

Figure 3.6: Dynamic tuning progress until 3000 time steps for the straight channel problem on Anantham

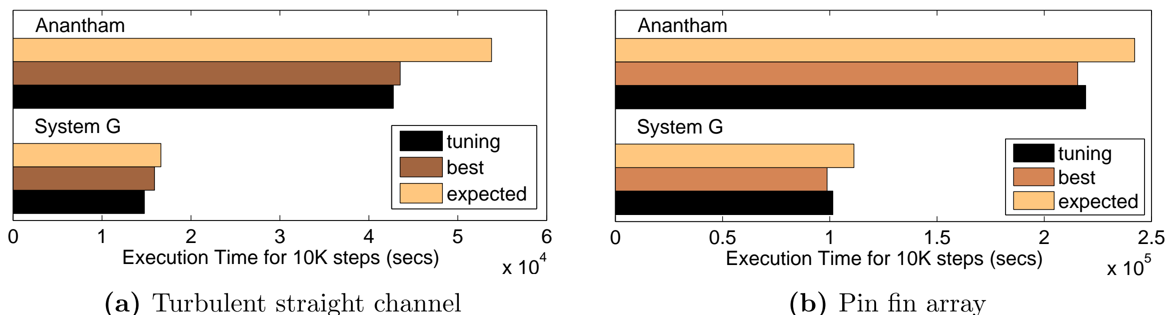


(a) Elapsed time between every 50 steps



(b) Cumulative elapsed time at every 50 steps

Figure 3.7: Dynamic tuning progress until 3000 time steps for the pin fin array problem on System G



(a) Turbulent straight channel **(b)** Pin fin array

Figure 3.8: Performance comparison of the GenIDLEST code for 10000 time steps

by 13% and 'best' by 8% on System G. Although the four configurations are only a small fraction of the whole search space, considering that they are practical samples, the results that our dynamic tuning method outperformed even 'best' cases show its effectiveness. For the pin fin problem, our method performed better than 'expected' by 10% but slightly worse than 'best' by 2% on Anantham, which is coincidentally the same on System G also. Further investigations of adaptation strategies taking into account the specificities of the domain being solved could lead to greater savings.

3.5.1 Tuning Cost

The runtime function call interception overhead incurred by ACC is small, amounting to less than 140 CPU cycles per intercepted call [39]. In addition, the piggybacking technique to intercept and compose at global operations in parallel MPI programs using ACC is reported to cause almost negligible overhead [55], which also leads to small tuning overhead in our experiments where we intercept only a few dozens of calls in the beginning of the simulations to implement our dynamic tuning strategy. Furthermore, we measured the search cost from exploring new parameter values to be very small, ranging from 0.4% (pin fin array problem on Anantham) to 0.9% (pin fin array problem on System G), where we counted those exploration steps that consumed more time than the previous 50 time steps.

3.6 Related Work

3.6.1 Auto-tuning

Auto-tuning techniques aim to generate automatically optimized numerical libraries to match the underlying hardware architecture. Model-driven auto-tuning uses analytical models for programs and machine architectures to perform efficient compiler transformations over program control structures such as loop blocking [57], loop unrolling, and software pipelining [58, 59]. While these model-driven compiler techniques are beneficial in that they make programs run faster in general without requiring any effort from the user, they may not always generate near-optimal code since compilers need to be general-purpose to be applicable to all kinds of programs, leading to simplified, general abstractions of the underlying processor

architectures, which limits accuracy in supporting special math kernels. In addition, the compiler models used may not be up-to-date with the newest hardware.

Empirical auto-tuning techniques, in contrast, parameterize the code for a given algorithm over the parameter search space and benchmarks each variant on a given platform to determine the best possible algorithmic option. For example, ATLAS [52] generates automatically tuned dense linear algebra routines and PHiPAC [60] generates parameterized code to produce highly optimized linear algebra routines such as matrix multiplication in ANSI C by searching for optimal block sizes to match the memory hierarchy of a given system.

While both PHiPAC and ATLAS take static tuning approach that performs optimization at the installation time, OSKI (Optimized Kernel Sparse Interface) [53], a library-based approach for auto-tuning of sparse matrix kernels, takes into account runtime performance behavior also. In addition to benchmarking blocking operations upon installation, OSKI employs runtime tuning in part by sampling a portion of the matrix under consideration and then estimating the performance for a given problem using the benchmark data. The runtime tuning cost can be non-trivial, up to 40× for sparse matrix-vector multiplication (SpMV), although it can be amortized with sufficiently many calls to the same operation as in iterative solvers.

In contrast to empirical tuning approaches, the main objective of our method is to support application programmers to separately reason about domain-specific dynamic tuning strategies and implement them onto existing programs, thus allowing for collective consideration of various runtime factors such as input problem attributes and parallel performance behavior of a whole execution environment.

3.6.2 Computational Steering

While our method is similar to Computational Steering (see Section 2.7) in that both target dynamic tuning of algorithmic parameters, our method supports the design and implementation of tuning strategies at the application composition phase, so that at runtime the tuning operations are automatically performed without visualization or interactive feedback system support.

3.7 Summary

The developed compositional approach addresses tuning as a separate concern in software development processes and allows one to separately design and implement dynamic strategies for scientific programs whose domain-specific algorithms are not easily supported by automatically tuned libraries. Unlike most auto-tuning techniques that consider only architectural characteristics of a single machine, the dynamic tuning method includes runtime factors as well such as input size and parallel performance behavior of a given execution platform, thereby enabling optimization of those parameters whose values cannot be determined *a priori* before application launch. Since performing exhaustive search at runtime is not viable with intolerable overhead for a large search space, design of a dynamic search strategy requires trade-off between search space and performance overhead; as demonstrated

the developed method was able to achieve upto 26% performance improvement in a practical scientific code with small tuning costs.

Chapter 4

Reusable Adaptation Patterns Implementation

4.1 Introduction

The execution model in the majority of computing domains has been consistently becoming more dynamic. In a dynamic execution model, the exact execution steps are determined only at runtime, as determined by input parameters and resource allocation. A significant portion of enterprise software, for example, is written in managed languages such as Java and C#. These languages not only dispatch methods dynamically to support polymorphism, but also heavily rely on dynamic class loading and Just-in-Time compilation. The default execution semantics is frequently adapted by means of Aspect-Oriented Programming [15], a programming paradigm that provides abstractions and tools to systematically augment or even completely redefine the semantics of method invocations and object construction. The AOP machinery is commonly applied dynamically, adapting the semantics of an application based on some runtime conditions.

Recently dynamic adaptation has been identified as capable of providing tangible benefits for scientific software [61, 62]. The traditional execution model of scientific applications—build, run, change, run anew—no longer provides the requisite flexibility required to accommodate the advanced needs of modern scientific applications. Such applications operate over ever-expanding data sets and require significant algorithmic sophistication to reach the needed performance levels.

Unfortunately, the traditional scientific software stack is tailored toward static execution. A significant portion of scientific applications are still written in Fortran, which despite all of its latest extensions still remains a glorified "formula translator" offering few facilities to support any execution dynamicity. The execution path of a typical scientific application is predetermined at compile time and rarely changes in response to any runtime events. To the best of our knowledge, no mainstream AOP extension has ever been developed for Fortran.

Thus to meet the need for dynamic adaptation, scientific programmers resort to manually introducing esoteric solutions that are often neither maintainable nor reusable. Although such solutions are recurring, their non-systematic implementation practices incur a significant software maintenance burden. Therefore, there is a great potential benefit in implementing

such dynamic adaption patterns more systematically and taking advantage of the state-of-the-art tools and techniques created for that purpose.

This work develops a solution to the problem outlined above by adapting Fortran programs by means of AOP, representing common adaptation patterns of scientific computing as reusable aspects. In lieu of a viable aspect extension for Fortran, our approach leverages the capabilities of AspectC++ [22], a popular C++ AOP extension. Thus, while a scientific programmer can continue maintaining the core functionality of a scientific application in Fortran, the adaptation logic is implemented in AspectC++ and automatically weaved with the original Fortran code.

The developed approach provides two main benefits. First, since Fortran remains the lingua franca of scientific computing, programmers can continue to develop and maintain their applications in this language. Second, all the adaptability functionality is implemented in AspectC++, which supports advanced software modularization and reusability principles through inheritance and abstraction. Since AOP is rapidly becoming an integral programming methodology in industrial software development, using a mainstream AOP language extension vastly increases the number of programmers who can maintain and evolve the added adaptation functionality.

To demonstrate that the approach is general and can benefit a substantial portion of scientific applications, this work has expressed a core set of common adaptation patterns as AspectC++ `abstract` aspects. By subclassing and concretely implementing these aspects, programmers can easily put in place sophisticated application-specific adaptation scenarios for scientific applications.

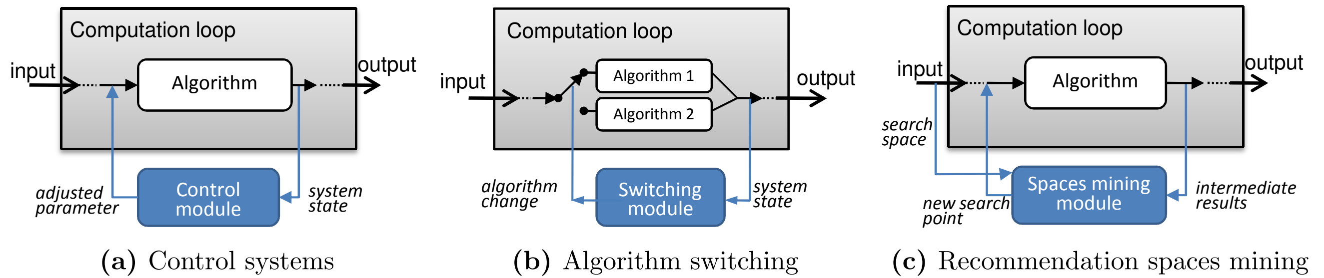
This work reports on the experiences of applying these scenarios to a real world scientific application—a suite of computational fluid dynamics applications. The resulting implementation shares identical performance characteristics with the original, non-reusable version that uses a special-purpose library to introduce the adaptation functionality. The aspect-oriented version, however, is more concise. On average, using inheritance reduced the amount of uncommented lines of hand-written code by as much as 27%. Thus, with the approach, the required adaptability functionality can be implemented more concisely, making it easier to maintain and reuse.

Based on these results, this work makes the following contributions:

- *An approach to rejuvenating scientific applications*: Adapting scientific applications provides efficiency, stability, or increased accuracy advantages. Our approach provides a systematic method to adapt scientific programs written in Fortran, thus allowing them to benefit from the mentioned adaptation advantages.

- *An approach to reusing adaptation code*: By expressing recurring adaptation patterns as abstract aspects that can be extended, our approach provides a reusable and customizable library of adaptations that can be used by different scientific applications.

- *Democratizing the writing of scientific adaptation functionality*: By exposing the adaptation functionality as standard AOP code, our approach increases the population of programmers who can write and maintain such code. Thus, while adapting scientific applications still requires expertise in the scientific domain at hand, implementing the



(a) Control systems **(b)** Algorithm switching **(c)** Recommendation spaces mining

Figure 4.1: Adaptivity schemas

functionality no longer requires intimate knowledge of the intricacies of Fortran. This is because adaptation functionality is introduced through AOP.

The rest of this paper is organized as follows. Section 4.2 summarizes common algorithmic level adaptation patterns in scientific computing. Section 4.3 describes our systematic method for applying aspect-oriented abstractions to Fortran code. Section 4.4 describes how we implemented adaptation patterns as reusable aspects and applied them to a realistic scientific application. Section 4.5 evaluates the software engineering benefits of our approach. Section 4.6 compares our approach to the related state of the art, and Section 4.7 presents concluding remarks.

4.2 Adaptation Patterns in Scientific Computing

In this section, we describe *adaptivity schemas*, common adaptation patterns of scientific computing, whose aspect-oriented implementation we describe in the next section.

4.2.1 Overview of Adaptivity Schemas

Adaptivity schemas [24] codify common adaptation patterns that occur in modern scientific applications. These patterns specify the scenarios under which the execution of scientific codes can benefit from being adapted dynamically. We demonstrate the concept of adaptivity schemas by describing three realistic use cases.

Control Systems

A Control Systems schema controls the algorithm of a scientific computation whose execution behavior can be affected by configuring the algorithm's parameters. The control system of such a computation can be realized through adaptation that adjusts the parameters to better match the dynamic characteristics of the computational progress. For example, Hovland and Heath [63] demonstrate how the relaxation parameter of the Successive Over-Relaxation (SOR) algorithm can be controlled through automatic differentiation. Figure 4.1a shows the schematic view of the Control Systems adaptivity schema.

Algorithm Switching

An Algorithm Switching schema describes those scenarios when the algorithm in place turns out to be inadequate to meet the requirements; the problem is then solved by dynamically switching to an equivalent algorithm. Switching algorithms can ensure greater accuracy or efficiency whenever numerical or physical properties of the computation in progress change. For example, the LSODE [64] solver, used in ODE systems, keeps its computation stable by switching between stiff and non-stiff methods over the region of integration. Hardwiring the switching procedure, however, often leads to using a conservative implementation as a means of preventing thrashing between the two categories of algorithms. A more flexible adaptivity implementation can take multiple runtime conditions into consideration when switching algorithms, thereby achieving greater computational stability without incurring the risk of thrashing.

Active Mining of Recommendation Spaces

An Active Mining of Recommendation Spaces schema describe those scenarios when adjusting algorithmic parameters dynamically can achieve greater levels of stability, efficiency, or accuracy. Choosing "sufficient" values heuristically may lead to sub-optimal results for different execution platforms. As the problem to be solved becomes more complex, the search space of algorithmic parameters can increase significantly to accommodate a greater number of processing units. This, in turn, can also negatively affect the accuracy of the resulting computation. The inadequacy of manual tuning and searching approaches for large search spaces motivates automated search and recommendation mechanisms. The active mining of recommendation spaces schema can selectively sample the parameter search space by analyzing the observed results, recommend a new set of parametric choices to be used in next loop iterations of computation, and keep repeating these steps until a desired functional is minimized.

4.2.2 Structural Characteristics of Scientific Programs

Since most execution time is spent on loops, they are a prime target of compiler optimization or parallelization techniques [65]. Adapting scientific programs also focuses on loops in most iterative computations—typically the end of a loop exhibits stable system state and consistent intermediate results. Hence, by placing adaptation code at the end of a loop, coherent results can be assessed without disturbing the ongoing computation. Furthermore, since parallel scientific programs typically synchronize concurrent execution at the end of a loop, an adaptation can reuse the barriers to achieve synchrony.

4.3 Adapting Fortran Programs via C++ Aspects

A typical scientific application is written in Fortran and uses the Message Passing Interface (MPI) [66]. This standard for programming distributed memory systems entails the SPMD (Single Program, Multiple Data) style, with all processes executing the same program with

different data. Aspect Oriented Programming provides powerful abstractions for implementing and applying the Adaptivity Schemas described above. Unfortunately, there is no AOP extension developed for Fortran. In the following, we describe the approach we developed that makes it possible to implement adaptivity schemas in AspectC++ and apply them to extant Fortran applications.

Using AOP provides two advantages. First, the adaptive functionality is implemented externally to the main code base and introduced at compile time, so that the Fortran and AspectC++ functionality can be maintained independently. Second, the language facilities of AOP provide greater opportunities for code reuse by means of aspect inheritance.

4.3.1 Integrating Fortran with AspectC++

Our approach leverages the binary compatibility between imperative languages compiled to the executable linkage format (ELF). What this entails is that functions in all imperative languages are compiled to interchangeable binary representations, as long as they use compatible data types for their parameters. Although there are differences in how advanced language features are implemented, the implementations of base features look identical at the binary level. For example, a Fortran function `foo` taking an `INTEGER` parameter is compiled identically to a C++ `static` function `foo` taking a pointer `int` parameter, with the only difference in how the compiled methods are named (i.e., Fortran methods are typically compiled to end with an underscore).

To work with AspectC++, Fortran code needs C/C++ equivalent, as AspectC++ uses source-to-source translation for weaving. We expose as such C++ equivalents only those portions of Fortran code that need to directly interface with aspects, specifically AspectC++ pointcuts. To expose function entry and exit pointcuts, we automatically (see Subsection 4.3.3) generate C wrappers, with wrapper C functions having the compatible signatures with the wrapped Fortran functions. AspectC++ can then add functionality (i.e., advice) to the original Fortran programs by means of the `execution` pointcuts.

4.3.2 Function Call Redirection

To redirect the invocations of the original Fortran functions to call the corresponding C wrappers instead, we use function call interception, a common technique with several implementations [34, 39]. Specifically, we use link-time wrapping, which is commonly supported on most Unix-based systems.

Link-time wrapping is provided by the system linker and wraps a function by changing its *symbol name*. At *program link time*, the linker globally renames the wrapped function, but the programmer is responsible for implementing the wrapper function. For example, the GNU linker, when passed the option '–wrap foo' to wrap the function `foo`, substitutes `__wrap_foo` for the `foo` references to generate the output. The linker also creates the `__real_foo` symbol for the original `foo` function, which can be used by the programmer to call `foo`.

An alternative implementation can use LD_PRELOAD, an environment variable that specifies dynamic libraries to be loaded into an application's address space early at load time, so that the system dynamic linker looks up the definitions for unresolved symbols

```
! Only the ITSOR subroutine signature is shown
SUBROUTINE ITSOR (NN,IA,JA,A,RHS,U,WK)
  INTEGER IA(1),JA(1),NN
  DOUBLE PRECISION A(1),RHS(NN),U(NN),WK(NN)
  ...

END

/* C itsor_wrapper.c */
#ifdef  __cplusplus
extern "C" {
#endif
// wrap the real ITSOR routine
void __wrap_itsor_ (int *nn, int *ia, int *ja,
  double *a, double *rhs, double *u, double *wk)
{
  return __real_itsor_(nn,ia,ja,a,rhs,u,wk);
}
#ifdef  __cplusplus
}
#endif
```

Figure 4.2: The `ITSOR` subroutine and its C wrapper code generated by our wrapper generator

(e.g., externally declared functions) However, that implementation works only with shared libraries, thus limiting its applicability.

4.3.3 Generating Fortran to C Wrappers

To implement a Fortran to C wrapper code generator, we extended F2PY [67], a Fortran to Python interface generator. The generated C code wraps Fortran functions/subroutines using the link-time wrap method. Essentially, in generating a C wrapper such that its signature matches that of a wrapped Fortran function/subroutine, our wrapper generator converts each Fortran argument type with its corresponding C type (e.g., Fortran `real` to C `float`) and uses pointer types to reflect Fortran's pass-by-reference parameter passing convention. We also exploit the fact that most Fortran compilers (e.g., GNU, Intel, and IBM compilers) on Unix-based systems support mixed-language programming by appending an underscore to function names when exporting symbols to linkers; our wrapper generator creates function names accordingly.

Figure 4.2 shows an example Fortran subroutine code[1] and its C wrapper generated by our wrapper generator. Here, the wrapper function, `__wrap_itsor_`, simply returns by making a call to the wrapped Fortran function, `ITSOR`, through its actual symbol name, `__real_itsor_`. The surrounding `extern "C"` linkage macro makes the C wrapper callable in C++ code, unaffected by C++ name mangling.

Figure 4.3 illustrates the overall structure of our approach that weaves AspectC++ aspects with extant Fortran scientific applications. First, the needed pointcuts in a Fortran

[1] `ITSOR` in ITPACK, http://rene.ma.utexas.edu/CNA/ITPACK.

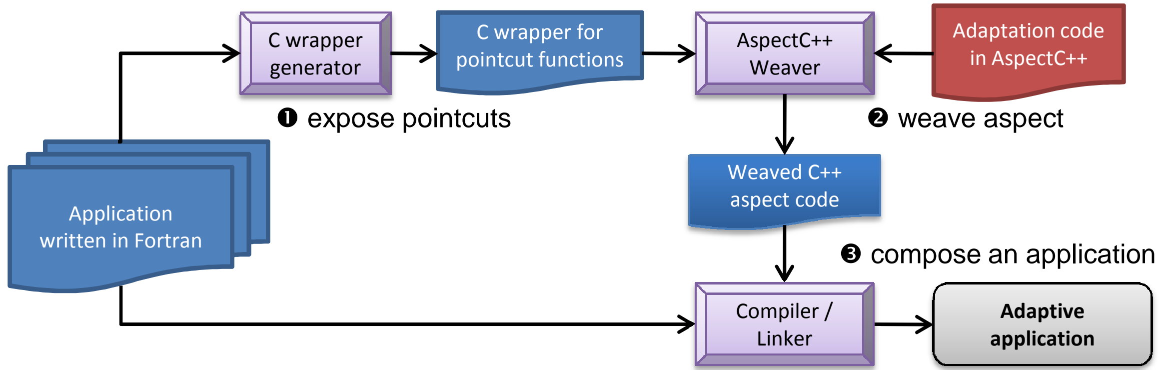


Figure 4.3: Weaving adaptation aspect code through AspectC++ by exposing Fortran functions in C wrappers

program are exposed as C functions through link-time wrapping and automated wrapper generation. The adaptation code expressed as AspectC++ advice structures is then weaved at the exposed pointcuts. Finally, all the weaved code is compiled and linked together with the original Fortran code, thus enhancing the original Fortran program with adaptive behavior.

4.4 Implementing Adaptivity Schemas in AspectC++

In this section, we describe our implementation of adaptivity schemas in AspectC++. Each implementation is showcased by an application to an HPC program called GenIDLEST [54], a computational fluid dynamics (CFD) simulation code written in Fortran 90 with MPI to solve the time-dependent incompressible Navier-Stokes and energy equations. The applications show how a Fortran scientific program can be adapted by weaved adaptation code to enhance its capabilities in various aspects of simulation such as stability, accuracy, and performance.

4.4.1 Obtaining Execution Environment Information

Most of the time, a parallel adaptation code needs runtime execution information, such as the size of the execution environment on which the application runs and the process ID. The adaptation code can use the information in performing its adaptation logic in such a way that the same code can make individual process behave differently depending on the process' unique status in the execution environment. In order to obtain the information at runtime, the adaptation code needs to interpose itself after initialization of the parallel environment is completed and perform such operations as to access the environment's information.

Figure 4.4 shows a baseline AspectC++ implementation for obtaining the MPI execution environment information. For an adaptation code to determine its rank (`myRank`) and the number of all processes (`numProcs`) when executed as an MPI application, it weaves an `after` advice at the Fortran MPI initialization function, `mpi_init`, exposed via its C wrapper, so that the `getParEnvInfo` function is executed to fetch the necessary information. The virtually declared `getParEnvInfo` can be overridden to include other necessary operations to obtain

```
protected:
  // parallel/MPI environment info
  int myRank;
  int numProcs;
public:
  // intercept parallel environment init function
  pointcut parInitFunc() = "% __wrap_mpi_init_(...)";
  // get the environment info
  virtual void getParEnvInfo() {
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size (MPI_COMM_WORLD, &numProcs);
  }
  // obtain parallel/MPI exec env info
  advice execution(parInitFunc()) : after() {
    getParEnvInfo();
  }
```

Figure 4.4: AspectC++ code for obtaining MPI execution environment information

extra application-specific information about the execution environment. For example, if an application uses multiple groups to organize tasks among the participating processes, the adaptation code can gain such grouping information through the MPI group and communicator routines to correctly perform its adaptation logic based on its status in the group. In some of our work, we use `getParEnvInfo` to execute application-specific initializations.

In the presentation of our work that follows, the adaptation pattern implementations assume the code in Figure 4.4.

4.4.2 Control Systems Schema

Figure 4.5 shows the AspectC++ implementation of the control systems schema. The `checkSystemState` virtual pointcut, which is to be concretely specified by subclasses, designates an interface through which a desired portion of the dynamic computation states can be retrieved. For Fortran programs we are targeting in this paper, this pointcut expresses the C wrappers that expose Fortran functions/subroutines selectively chosen by the programmer to access system state.

The `after` advice is used at the `checkSystemState` pointcut to interpose adaptation code, where parameters of the used algorithm are adjusted with regard to changes in program state. The state information is fetched through the AspectC++ `tjp->result()` API after the execution of the designated Fortran pointcut function, and is then passed to `adjustParam`. The virtually declared `adjustParam` function is to be implemented by extending subclasses that embody application-specific adaptation strategies to control program state.

As an application of the AspectC++ implementation of the control systems schema, we extend the schema code to implement a simulation stability control logic, which is similar to our previous work [55] where we used a composition tool called Adaptive Code Collage (formerly called Invoke) to plug separately written adaptation code into GenIDLEST without modifying the original source code. As the stability of the GenIDLEST simulation depends on the time step size used, we monitor Courant-Friedrich-Levi (CFL) numbers to check if

```
aspect ControlSystems {
public:
  // pointcut at system state function
  pointcut virtual checkSystemState() = 0;
  // adjust system parameters
  virtual void adjustParam(void* state) = 0;
  // adjust parameters to adapt to system state
  advice execution(checkSystemState()) : after() {
    void* state = tjp->result();
    adjustParam(state);
  }
};
```

Figure 4.5: Control systems schema implementation in AspectC++

the simulation is proceeding towards convergence or is becoming unstable. The adaptation code automatically adjusts the time step parameter to allow the computation to proceed in a stable manner.

Figure 4.6 shows the time step control code for GenIDLEST, in which the `ControlSystems` aspect class is extended by `TimestepControl`. The Fortran function specified as a pointcut is `get_convcfl`, which returns a global convection CFL number. Inside `adjustParam`, the time step parameter is accessed through `get_dt` and is updated with a new value through `set_dt`. The adaptive logic employed here is a simple multiplicative increase/decrease algorithm with upper (`CFL_U_THRESHOLD`) and lower (`CFL_L_THRESHOLD`) threshold values for the CFL number, such that, if the observed CFL number becomes out of the bounds defined by the thresholds, the time step value is increased or decreased by a preset factor (`DT_DAMPING_FACTOR`).

4.4.3 Algorithm Switching Schema

Most scientific programs use an integer parameter value to specify an algorithmic option to be used in computation. For example, the LSODE solver accepts from the user an integer value for a numerical method among a set of stiff and non-stiff algorithms for a given problem. Hence, this allows a program component to switch to a different algorithmic option by changing the parameter. We implement the algorithm switching schema based on this convention.

Figure 4.7 shows our AspectC++ implementation of the schema expressed in the `AlgoSwitching` aspect class. The virtual `globalComm` pointcut needs to be specified by extending classes such that some global operation is designated as a place for aspect code insertion. Since algorithm switching needs to be performed synchronously across all the processes to avoid races that can cause inconsistent results, it is important to correctly define this pointcut in subclasses. Functions that execute global communications while placed near the computation loop would be a good target for this pointcut as we described in Section 4.2.

The `recommendSwitching` function returns a `bool` value for dynamic adaptive decisions about algorithm switching, which, depending on applications, can either be automated by an adaptive procedure or be initiated by the domain expert based on analysis of observed results. `getNewMethod` returns an integer that specifies an algorithmic option for switching. Subclasses

```
#define CFL_THRESHOLD 0.5
#define DT_DAMPING_FACTOR 0.5

aspect TimestepControl : public ControlSystems {
  // CFD convection CFL
  pointcut checkSystemState() =
           "% __wrap_get_convcfl_(...)";

  void adjustParam(void* state) {
    double dt, new_dt;
    dt = get_dt();
    // control timestep to keep CFL within bounds
    if (*(double*)state >= CFL_THRESHOLD) {
      new_dt = dt * DT_DAMPING_FACTOR;
      set_dt(&new_dt);
    }
    else if (*(double*)state < CFL_THRESHOLD/2.0) {
      new_dt = dt * (1.0 + DT_DAMPING_FACTOR);
      set_dt(&new_dt);
    }
  }
};
```

Figure 4.6: Timestep adaptation aspect to improve the stability of GenIDLEST simulations

need to provide a concrete definition for each of these functions.

The `after` advice executes switching operations after the `globalComm` pointcut function completes. Here, the root process makes the switching decision by calling `recommendSwitching` and `getNewMethod`, and the decision is then broadcast to all the processes by the `switchMethod` function through the `MPI_Bcast` global communication. Since `MPI_Bcast` synchronizes the execution of all the processes to perform correct switching, it can cause overhead in application performance. However, as the operation "piggybacks" onto the existing global communication specified as `globalComm` so that these separate barriers are placed close together, the combined overhead becomes smaller and the potential performance slowdown can be mitigated. Subclasses are expected to override `switchMethod` and include application-specific operations necessary to completely realize algorithm switching, since the baseline implementation of `switchMethod` only communicates the switching decision among processes.

As an application of the algorithm switching schema to GenIDLEST, we implemented flow model switching to improve the simulation accuracy. In CFD simulations, the predicted heat transfer and flow characteristics depend on the selection of the appropriate flow model such as laminar or turbulent models. Since the physics of the simulated flow cannot be known *a priori*, improper choice of flow models may cause inaccurate results, in which case stopping the current execution and resuming the simulation with a correct model is required. To avoid such cases and make a simulation proceed without stop, our application implements flow model switching based on the `AlgoSwitching` aspect code.

Figure 4.8 shows the flow model switching implementation in AspectC++. The adaptation logic is similar to our previous work in [55], where a switching decision is dynamically

```
aspect AlgoSwitching {
public:
  // intercept global communication for advice
  pointcut virtual globalComm() = 0;
  // recommend switching based on some metric
  virtual bool recommendSwitching() = 0;
  // return a new algorithmic option
  virtual int getNewMethod() = 0;
  // perform switching
  virtual void switchMethod(int method) {
    MPI_Bcast(&method,1,MPI_INT,0,MPI_COMM_WORLD);
  }

  advice execution(globalComm()) : after() {
    int newMethod;
    // root process initiates switching
    if (myRank == 0) {
      if (recommendSwitching()) {
        newMethod = getNewMethod();
      }
    }
    switchMethod(newMethod);
  }
};
```

Figure 4.7: Algorithm switching schema implementation in AspectC++

made by the user with domain knowledge and Unix signals are used to initiate and effect model switching onto running MPI processes. For the pointcut function to interpose adaptation operations, we use the `calc_cfl` subroutine located at the end of the time integration loop, which uses MPI reduction operations to calculate CFL numbers. Base class's `getParEnvInfo` is overridden to install a Unix signal handler, which sets the `user_stop` flag at the user's request sent via a signal, so that `recommendSwitching` returns `true` at the next iteration of time integration. `getNewMethod` takes the user's switching decision through a user interface provided by the root process and `switchMethod` performs switching by setting the model parameter with the value passed from `getNewMethod`. After switching is complete, `user_stop` is reset to `false`.

4.4.4 Active Mining of Recommendation Spaces Schema

Figure 4.9 shows our AspectC++ implementation of the recommendation spaces mining schema. The aspect is designed to execute each step of *point selection in search space, exploration of a selected point, evaluation, and search space update* at the loop end in consecutive iterations. The `after` advice for `parInitFunc` executes initialization for search and exploration such as obtaining search space information in the advice code for `parInitFunc`. `DTYPE` and `MPI_DTYPE` are macros to support multiple data types (e.g., `int` and `double`) of parameters that define search space, and are required to be defined with a specific type in subclasses.

The `after` advice placed at `globalComm` begins exploration of search space by selecting a

```
// code for Unix signal (SIGUSR1) handling
bool user_stop = false;
static void sigusr1_handler(int sig);
static void install_sigusr1_handler();

aspect FlowModelSwitching : public AlgoSwitching {
public:
  // use calc_cfl() to execute adaptation
  pointcut globalComm() = "% __wrap_calc_cfl_(...)";
  // override to install a signal handler
  void getParEnvInfo() {
    AlgoSwitching::getParEnvInfo();
    install_sig_handler();  // SIGUSR1 handler
  }
  // set user_stop at user's request
  bool recommendSwitching() {
    bool recommend = false;
    if (user_stop) recommend = true;
    return recommend;
  }
  // accept switching decision via UI
  int getNewMethod() {
    int newMethod;
    // user interface code for switching decision
    ...
    return newMethod;
  }
  // effect flow model switching
  void switchMethod(int newMethod) {
    AlgoSwitching::switchMethod(newMethod);
    setNewMethod(&newMethod);
    user_stop = false;
  }
};
```

Figure 4.8: Flow model switching aspect to improve the accuracy of GenIDLEST simulations

new point from a given space (`getSearchPnt`). The virtually declared `beginExplore` needs to be concretely specified in subclasses such that a running computation is updated to use newly selected parameter values in next iterations. The computational progress and its properties in the exploration step are regularly checked by `checkExplore` at the loop end, which needs to be specified by subclasses to decide when to stop exploration. After exploration completes, the root process compares the explored point with the previous point, determines which to use for the ongoing computation (`evalExplore`) based on some metric such as execution time, and broadcasts its decision. Finally, the search space is updated with the exploration result and a new round of search begins in the next iteration.

As an application of the mining schema aspect to GenIDLEST, we implemented dynamic parameter tuning of algorithmic parameters in AspectC++, which is also similar to our previous work [68] where we used Adaptive Code Collage (ACC). In this application, a 3-dimensional integer parameter that represents the size of the data structure used in the

```
aspect Mining {
protected:
  bool srchComplete, explrComplete;
  DTYPE *srchPnt, *curPnt;
  unsigned int count, dim;
public:
  pointcut virtual parInitFunc() = "% __wrap_mpi_init_(...)";
  pointcut virtual globalComm() = 0;
  // functions for managing search space
  virtual DTYPE * getSpace() = 0;
  virtual DTYPE * getSearchPnt() = 0;
  virtual bool updateSpace(DTYPE * pnt) = 0;
  // functions for performing exploration
  virtual void beginExplore() = 0;
  virtual bool checkExplore() = 0;
  virtual bool evalExplore() = 0;

  ... // perform initialization
  advice execution(parInitFunc()) : after() {
    getParEnvInfo();
    count = 0;
    explrComplete = true;
    srchComplete = false;
    curPnt = getSpace();
  }

  // mine search space at every iteration
  advice execution(globalComm()) : after() {
    count++; if (srchComplete) return;

    if (explrComplete) {
      srchPnt = getSearchPnt();
      explrComplete = false;
      // use new parameters in next iterations
      beginExplore();
    } else {
      explrComplete = checkExplore();
      if (!explrComplete) return;

      DTYPE *dPnt = curPnt;
      // root process makes decision
      if (myRank==0) {
        if (evalExplore()) dPnt = srchPnt;
      }
      MPI_Bcast(dPnt,dim,MPI_DTYPE,0,MPI_COMM_WORLD);
      curPnt = dPnt;
      if (!updateSpace(srchPnt)) srchComplete = true;
    }
  }
};
```

Figure 4.9: Mining of spaces schema implementation in AspectC++

GenIDLEST preconditioning code, called cache sub-blocks, is dynamically tuned through a staged optimization procedure to match the memory hierarchy of a given execution platform

Figure 4.10 shows a part of the tuning aspect implementation, where we list only the most relevant part of the tuning logic. To find a candidate point in the cache sub-block parameter space, `getSearchPnt` searches in each of the 3-dimensional space that is bounded by current minimum and maximum values. Although not entirely shown, in order to balance the trade-off between tuning cost and application performance, the actual implementation uses a set of optimization schemes to effectively reduce the search space size during the process.

The `checkExplore` function uses a 4-stage procedure to evaluate a new cache sub-block parameter value (i.e., the search point being explored) and compare with a previous value using execution times spent to complete a preset number of loop iterations. Based on measured timings for each of the new and previous parameter values, `evalExplore` decides which one to use in the running computation and the search space is updated accordingly.

4.5 Evaluation

We evaluate our AspectC++ adaptation aspects with respect to code reusability, software complexity, and performance overhead, and compare our implementations with hand-written code. Specifically, we compare our AspectC++ implementation with the manually written adaptation code described in our previous work [55, 68].

4.5.1 Reusability

A desirable software design objective is code reusabiltiy, which allows using the same code fragments in multiple scenarios either within the same application or across different applications. The ability to reuse code improves programmer productivity, as the programmer does not have to implement the same functionality multiple times. This, in turn, leads to a smaller code size, which reduces the maintenance burden and the risks of introducing software defects.

In object-oriented programming, an important technique to promote code reusability is class inheritance. Common functionality is encapsulated in a base class that is extended by subclasses which add only the unique functionality. In AOP, aspects can inherit from each other. In our GenIDLEST adaptation implementation, we use inheritance to extend adaptivity schema aspects, thereby reducing the total size of the adaptive code.

TABLE 4.1 compares the amount of uncommented lines of source code (ULOC) written by a programmer between the AspectC++ and hand-coded implementations. 'aux' represents auxiliary code that is not relevant to an adaptation logic implementation (designated as 'logic' in the table), such as header includes, helper functions, and linkage macros to resolve name mangling between Fortran and C/C++. The hand-coded implementations also need to use the ACC framework's APIs to setup the introduction of adaptation code to GenIDLEST.

The AspectC++ versions take less code to implement than the hand-coded ones in all cases. The code reduction ranges between 15% for the most complex dynamic tuning adap-

```
aspect ParamsTuning : public Mining {
private:
  bool blkXdone, blkYdone, blkZdone;
  double exeTm, curBlkTm1, curBlkTm2, expBlkTm;
  int minXblk, maxXblk, ...;
public:
  int *getSearchPnt() {
    int *tmpNblk = new int[SPACEDIM];

    if (!blkXdone) { ... //search in X direction }
    else if (blkXdone && !blkYdone) { ... //Y }

    ...   //set tmpNblk with new sub-block numbers
    return tmpNblk;
  }
  ...

  bool checkExplore() {
    bool exploreStatus = false;
    endTm = getTimeStamp(); exeTm = endTm-startTm;
    //4-stage procedure for dynamic tuning
    int stage = count%NSTAGE;
    switch (stage) {
    case 1: curBlkTm1 = exeTm; break;  //1st measure
    case 2: curBlkTm2 = exeTm;         //2nd measure
            setParams(srchPnt); //now set new params
            break;
    case 3: expBlkTm = exeTm; //new params exe time
            exploreStatus = true; break;
    default: break;
    }
    startTm  = getTimeStamp();
    return exploreStatus;
  }
};
```

Figure 4.10: Dynamic parameter tuning aspect to improve the performance of GenIDLEST simulations

tation and 40% for the simple time step control adaptation. The adaptive functionality for the auxiliary part can be implemented in fewer lines of code by using AspectC++ and our C wrapper generator. The hand-coded implementation also requires some hand-written code to properly instantiate the ACC framework.

In addition, AspectC++ implementations use fewer lines of hand-written code by inheriting schema aspects. As code becomes complex and its size grows, the table shows that the gain becomes smaller because the pointcuts defined in all adaptation implementations specify a limited number of join points such as loop ends. Therefore, application-specific adaptation schemes that involve multiple join points can benefit more from subclassing schema aspects.

	Hand-coded			AspectC++		
Adaptation	aux	logic	total	aux	logic	total (gain)
Timestep control	17	31	48	9	20	29 (40%)
Model switching	20	68	88	13	53	66 (25%)
Dynamic tuning	25	172	197	13	154	167 (15%)

Table 4.1: ULOC comparison of the GenIDLEST adaptation implementations between the hand-coded and AspectC++ implementations

	Hand-coded		AspectC++	
Adaptation	aux	logic (next max)	aux	logic (next max)
Timestep control	2	3 (0)	0	3 (0)
Model switching	2	5 (0)	2	3 (2)
Dynamic tuning	1	25 (6)	1	14 (13)

Table 4.2: Complexity comparison of the GenIDLEST adaptation implementations between the hand-coded and AspectC++ implementations using maximum MCC numbers

4.5.2 Software Complexity

AOP refactoring enables greater modularity of program components by modularizing cross-cutting concerns while preserving the external behavior. Thus AOP reduces software complexity and thereby increases maintainability and productivity. To measure the software complexity of our AOP-based adaptation implementations, we use McCabe Cyclomatic Complexity (MCC) [69], a metric indicative of the effort required to understand a codebase.

TABLE 4.2 compares maximum MCC numbers between the AspectC++ and hand-coded adaptation implementations. The 'next max' number is the MCC number of the function that shows the second biggest MCC number. For the time step control adaptation, the complexities of both implementations is the same. The auxiliary part in the hand-coded version has MCC of 2. For the flow model switching adaptation, the complexity of auxiliary code is the same in both implementations, as both use the Unix signal handling functions. However, the complexity of the adaptation logic code becomes reduced because of the organized structure inherited from the algorithm switching schema aspect code. This effect becomes significant in the dynamic tuning implementation, which is the most complex code of all adaptations. The maximum MCC number is greatly reduced from 25 in the hand-coded implementation to 14 in the AspectC++ version, while the second largest MCC number for the AspectC++ implementation is bigger than the hand-coded one. This is mostly because the hand-coded implementation intermingles codes, such as that for the timing and parameter space updating functionality, with many conditional statements. In contrast, the AspectC++ implementation follows the organized structure in the mining base class, thereby keeping the overall complexity balanced across different functions.

Cluster	**Original GenIDLEST**	**GenIDLEST with Aspects**		
		Timestep Control	Model Switching	Dynamic Tuning
Anantham	9441	9448 (.1%)	9473 (.3%)	9519 (.8%)
System G	4105	4110 (.1%)	4124 (.4%)	4134 (.7%)

Table 4.3: Execution time (seconds) and overhead measurements of the GenIDLEST adaptation implementations using a pin fin array problem for 500 time steps.

4.5.3 Performance Overhead

To measure the performance cost caused by imposing adaptation operations onto GenIDLEST, we perform GenIDLEST simulations on two cluster systems, called Anantham and System G, respectively. Each node of Anantham is a 64bit Linux (kernel version 2.6.9) machine with a 1.4GHz AMD Opteron 240 dual-core CPU and 1GB of memory, interconnected with 100Mbps Ethernet. The MPI runtime used is MPICH 2.1 with GCC 4.2.5. System G consists of dual-socket 2.8GHz Intel Xeon E5462 quad-core SMP machines interconnected with 40Gbps InfiniBand. The operating system on System G is the 64bit Linux 2.6.27 kernel and the MVAPICH 2.1 MPI system was used with GCC 4.3.2. A pin fin array was used as an example CFD problem, which is decomposed into 16 blocks such that each block is processed by one MPI process (i.e., the number of parallelism is 16). On Anantham, 8 nodes with two processors each were used, while on System G, 4 nodes with 4 processors each were used. In the experiments, each application-specific adaptation logic is disabled and only the base class operations are performed, so that the overhead caused only by the pattern implementations are measured.

TABLE 4.3 shows the total execution time it took to run the entire GenIDLEST simulation, using both the hand-written and AspectC++-based adaptation approaches. To extract the adaptation overhead, these execution times are compared to that of the original GenIDLEST program. Since the time step control adaptation is the simplest and does not use any global operations, its overhead is the smallest of all on both platforms. For adaptations that use more complex patterns that execute global operations, such as algorithm switching and mining, the incurred overhead grows. However, the performance cost of adaptation aspects is quite small, incurring 0.8% across the platforms. This overhead is comparable to that incurred by the hand-written versions implemented using the ACC framework.

4.6 Related Work

4.6.1 Multilingual Systems

Other approaches to integrating Fortran with C/C++ focused on language translation. There are Fortran 77 to C translators, such as `f2c` at `http://www.netlib.org/f2c`. Language interoperability tools include middleware such as Babel[70] and component technolo-

gies such as Common Component Architecture[71]. These technologies support multiple languages but require either specific API conformance or the use of special interface definition languages. These approaches are too heavy weight for the purposes of this work.

4.6.2 AOP for Scientific Computing

Although scientific computing was one of the initial application domains of AOP[72], the AOP methodologies and abstractions have not been deeply investigated in the scientific computing area. This is mostly due to the fact that scientific applications are typically written in Fortran or C/C++ for performance and scalability reasons, while a large body of the AOP research is based on Java-based implementations. However, it is encouraging that the overhead of Java (e.g., garbage collection overhead) is becoming acceptable for computationally intensive tasks with the increasing hardware parallelism [73], which can lead to broader recognition of sophisticated software engineering methodologies such as AOP in the scientific and high-performance computing domains. Harbulot et al. [74] tackles the code-tangling issue in parallel scientific programs, where computation code is intermingled with parallelization code in such a way that further software changes become difficult. Their work applies AOP refactoring to separate the parallelization concern in a scientific application into a single aspect, thus achieving modularity. Han et al. [75] showcases AOP applications to cluster computing software. Their work modularizes several additional functionalities for the MPI library, such as fault-tolerance and routing between heterogeneous clusters, into aspect code and uses AspectC++ to combine them with MPI, thereby creating an enhanced version of MPI.

4.6.3 AOP for Parallel Programming

Several AOP research work treat parallelization as a separate concern, so that a parallel version of an application can be generated from a serial code in a modular way by plugging in separately developed parallel aspect code through AOP frameworks. Sobral [76] uses AspectJ for incrementally developing parallel applications with serial Java programs. Harbulot and Gurd [77] develops a join point model and a compiler for recognizing loops, which are a prime target of parallelization, so that aspect code can be interposed at the loop level.

There are AOP frameworks that use annotations to express concurrency aspects, which is similar to the OpenMP model [78] that uses compiler directives to express parallelism. For example, both the JBOSS AOP framework [79] and recent versions of AspectJ [80] provide the `@Oneway` annotation to fire `void` methods in a separate thread that will run asynchronously in a task-parallel fashion. Cunha et al. [81] is similar to our work in that it presents reusable aspect-based implementations of a set of common concurrency patterns, such as futures, barrier, and synchronization. However, their work implements concurrency patterns on shared-memory platforms, while ours focuses on adaptation patterns in scientific computing on parallel platforms in a distributed-memory environment. Also, their aspect implementations based on AspectJ targets programs written in Java, a language ingrained with OO mechanisms already. In contrast, our work attempts to apply sophisticated OO abstractions to Fortran programs, in which OO mechanisms are rare, by selectively exposing the code as aspect pointcuts.

4.7 Conclusion

In this paper, we presented a novel approach to expressing recurring adaptation functionality patterns of scientific computing as reusable aspect-oriented code. Our approach uses cross-language adaptation implemented using code generation and an aspect library. We evaluated the software engineering benefits of our approach by obtaining the ULOC and cyclomatic complexity metrics from the original (hand-coded) and our (aspect-based) versions of a computational fluid dynamics scientific application. The results of the evaluation show that using aspects can reduce the amount of code needed to implement the adaptivity functionality by as much as 27% on average. We have also verified that using our approach does not incur an unreasonable performance overhead.

Overall, the software engineering benefits of our approach include improved maintainability, more structured design, and greater automation. Greater reusability enabled by our approach also allows scientific programmers to subclass the schema aspects provided by our library, thereby reducing the programming effort.

Chapter 5

Implementing Runtime Adaptation of Scientific Applications

5.1 Introduction

To adapt an application's runtime behavior requires specifying adaptation schemes, which can be categorized into *precisely defined schemes* and *loosely defined schemes*. Precisely defined schemes include a complete description, *before* application launch, of "where" (adaptation control points), "when" (conditions to trigger adaptation actions), "what" (adaptation targets such as variables or functions), and "how" (operations to execute) to change with regard to runtime program behavior. In contrast, loosely defined schemes consider highly dynamic adaptation scenarios in order to support the user's dynamic and unplanned adaptation decisions. This kind of adaptation schemes lack complete adaptation specifications: some part of adaptations are not specified clearly and adaptive decisions are deferred until at runtime. Adaptation decision in such cases can only be made by monitoring the application's dynamic progress, since, without actually running it, the user may have just a vague idea of how the program in consideration evolves.

Implementing loosely defined schemes requires support of high degree of execution control of an application due to the fact that adaptation operations are determined by the user's dynamic decisions. This kind of schemes require a running application to stop at the user's discretion, update its system state with user input, continue its execution followed by additional stops for possible future adaptations. Furthermore, these schemes often require an application to record and restore its state so that its computation can be resumed at an execution point which the user may think is interesting for performing experiments or simulations. Therefore, realizing loosely defined adaptations in a modular becomes complex. This is in contrast to implementing precisely defined schemes, where adaptation operations are automatically initiated by writing adaptive logic operations in a separate code and by plugging the code into an application.

In this work, we present a modular approach to implementing loosely defined adaptation schemes. We use a compositional framework to insert adaptation code into an existing application. To support the user's dynamic adaptation decisions, the adaptation code is inserted at a place where an application executes regularly such as the end of a simulation

loop. Instead of executing automated adaptation operations, the main functionality of the inserted adaptation code is to serve as a gateway to control the execution of an application. Whether to stop or continue the execution is guided by the user. Therefore, the inserted adaptation code is executed at regular interval and allows the user to check the application progress periodically. When adaptation is needed, the user can manipulate the execution control and initiate adaptations through the inserted adaptation code.

By using our approach, domain experts in scientific simulations can realize dynamic adaptation decisions and "what-if" scenarios at program runtime, so that they can better mimic what experimentalists do in a physical environment. We demonstrate the applicability of our approach through adaptive scenario implementations in modeling and simulating biological systems.

5.2 Adaptive Modeling of Cell Cycles

In search of better models or in exploration of new models, cell cycle modelers change the configuration of a given simulation to examine how the original system is perturbed to evolve into a new state. The course of action using conventional modeling tools that lack dynamic program adaptation functionalities is: 1) stop the simulation and save the state, 2) update the mathematical model (i.e., system of ODEs) to reflect the perturbation by changing model parameter values, 3) set the initial conditions of the ODEs with the saved state, and 4) restart the simulation. Repeating the procedures whenever configuration changes are desired is cumbersome for cell cycle modelers. Therefore, the ability to adapt simulations at runtime that does not require interruptions helps facilitate effective cell cycle modeling.

Adapting a simulation at runtime requires a dynamic change in the original simulation model quantified by a set of ordinary differential equations (ODEs). In particular, dynamic changes required for effecting perturbations can be loosely defined. For example, specific reactions to change in a simulated system can remain unspecified before simulation launch and be up to the modeler's runtime decisions.

In this section, we describe the benefits of dynamic adaptation in performing cell cycle simulations.

5.2.1 Effective Parameter Estimation

In a mathematical model that quantifies a biological reaction system, each reaction is modeled as an ODE with parameters that represent reaction rates. Parameter estimation plays a crucial role in modeling biological systems based on real experimental data. Dynamically adapting scientific software through loosely defined schemes allows for increased execution control and interactive feedbacks to a running computation, so that, based on runtime analysis of computational progress, the user can make adaptive decisions and manipulate ODE parameters immediately through feedback controls without stopping a running simulation. The user then can check how the changes in model parameters make the simulation better match experimental data. Thus, support of loosely defined schemes in biological simulations expedites effective parameter estimation.

5.2.2 Exploration of New Evolution Pathways

In addition to facilitating effective parameter estimation, adapting cell cycle simulations at runtime allows the user to explore new evolution pathways by manipulating the model parameters of a running simulation. Without having to stop a running computation, the user can dynamically simulate a new *protocol* – a sequence of operations performed in real experiments to develop a certain environment to suppress or catalyze specific biochemical reactions, such as raising the temperature or adding new materials to a system – to investigate how the simulated system evolves from a current condition. Hence, dynamic adaptation capabilities here can effectively facilitate new scientific discoveries.

5.2.3 Effective Simulation through Checkpointing and Rollback

Most conventional simulation tools are static in the sense that a simulation cannot be rolled back to past states once it is complete. As a result, the user has to finish a simulation and record its results whenever he or she finds the results interesting for future uses, which becomes time-consuming and error-prone as the simulated problem becomes complex and the number of simulations involved gets large. Through the increased execution control offered by the dynamic adaptation capabilities, a running computation can be adapted to pause to save its entire states on-the-fly and then resume its execution. By saving and restoring intermediate results and simulation configurations, the user can easily reuse past simulations of interest later in future simulations, thus being able to shorten the time-to-solution.

5.3 Implementation

To implement program adaptation in a modular way without directly modifying the original application source code, we use the Adaptive Code Collage (ACC) framework that is described in Section 2.4. ACC is a compositional framework with a set of function call interception APIs, through which every call to a function of interest is intercepted and program control is diverted to an associated *handler*, a piece of newly inserted code responsible for monitoring and modifying the target function's behavior. Therefore, by integrating separately written adaptation code with the original program through ACC, we essentially compose an adaptive application enhanced with newly added capabilities.

In this work, we implement dynamic adaptations using the PET (Parameter Estimation Toolkit) [82] software as an example. PET offers two phases of cell cycle modeling: parameter (reaction rate) estimation and simulation of the biochemical systems. Specifically, we implement adaptations over the Fortran code generated by PET in the simulation phase, for which PET uses the `lsodar` solver in ODEPACK [83], a collection of Fortran solvers for ODE systems.

5.3.1 Adaptation Control

Since the end of a simulation loop typically exhibits stable system state with consistent intermediate results, we specify the entry of the `lsodar` function as an adaptation control point to control the execution of the PET simulation. Thus, the `lsodar` calls in the PET source file

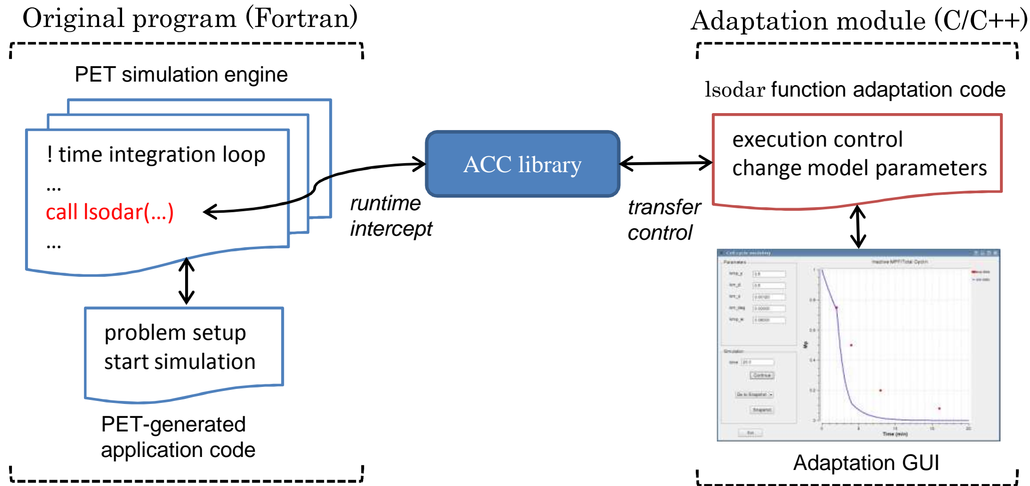


Figure 5.1: Dynamic adaptation implementation of PET cell cycle simulations using ACC

`lsodar_mod.f90` are intercepted by the ACC framework. We implement an adaptation scheme in a function called `lsodar_handler`, to which the intercepted `lsodar` calls are redirected by ACC.

The `lsodar_handler` function checks the time step at each iteration of the simulation loop and compares it with a stop time specified by the user through a GUI (described in the next subsection), so that the simulation is paused when the stop time is reached. While the application is stopped, the user can analyze the simulation progress, make adaptive decisions, and adapt the simulation by changing the values of the model parameters of interest.

To access the model parameters and variables of a simulated system, we reuse a set of `set/get` helper functions generated by PET such as `get/set_model_param` and `get/set_model_var`.

5.3.2 GUI for Runtime Simulation Adaptation

To better support cell cycle modelers in performing dynamic adaptations, we prototyped a GUI whereby the modeler can control the execution of simulated computations and change parameter values of a given model dynamically. We implement the GUI using `Qt` [84], a cross-platform UI development framework. The GUI runs in a separate thread from the main simulation engine using the `Pthread` APIs [85] in order to control the simulation execution separately from the GUI execution. We describe the GUI in detail with examples in the next section.

Figure 5.1 illustrates how we dynamically adapt PET cell cycle simulations through the ACC framework. A simulation program originally consists of the PET simulation engine and a PET-generated simulation driver, both written in Fortran. The adaptation program module includes the `lsodar` handler code and the GUI. The ACC framework combines a simulation program and the adaptation module to compose a simulator that realizes user-driven adaptation schemes.

JigCell ModelBuilder

ModelBuilder File Edit

Reactions | Functions | Rules | Compartments | Species | Parameters | Events | Units | Conservation Relations | Equations

Species	Equation
dM/dt	(km_d * (Dp + Dsp) * Mp) - ((kmp_y * Myt + kmp_w * Wp) * M) + (km_s) - (km_deg * M)
dMp/dt	- (km_d * (Dp + Dsp) * Mp) + (kmp_y * Myt + kmp_w * Wp) * M - (km_deg * Mp)
HalfMT	MT / 2.0
dD/dt	- ((kdp + kdp_m * M) * D) + (kd_p * Dp) - ((kdp2 + kdp2_c * Chk1) * D) + (kd_p2 * Dp2)
dDp/dt	(kdp + kdp_m * M) * D - (kd_p * Dp)
Mytp	-(Myt - MytT)
dMyt/dt	(ky * Mytp) - (kyp_m * M * Myt)
Dp2	-(D + Dp - DT)
dW/dt	- ((kwp + kwp_c * Chk1) * W) + (kw * Wp)
Wp	-(W - WT)
dlamin/dt	- (klp_m * M * lamin)
laminp	-(lamin - laminT)
NEB	Gtrunc(laminp,0.0,1.0)
dDsp/dt	- (kd_p * Dsp) + (kdp + kdp_m * M) * Ds
Ds	-(Dsp - DsT)

Figure 5.2: ODE system that models the cell cycles of frog-egg extracts

5.4 Experiments: Dynamic Adaptation of Cell Cycle Simulations

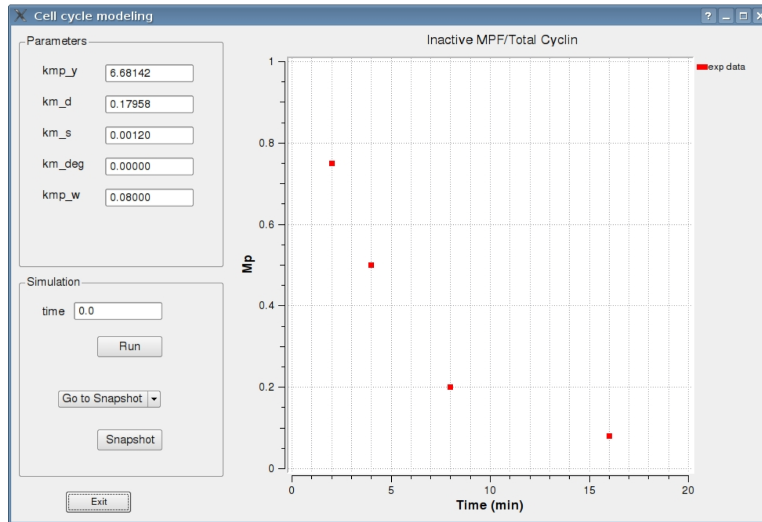
As an example simulation of cell cycles, we use the frog-egg extracts simulation [86]. Specifically, we apply the adaptation code and the GUI to the PET-generated Fortran code for simulating dephosphorylation of pre-MPF (mitosis-promoting factor) during mitosis, when *Cdc25* is active [87]. The biological system is modeled into a system of 15 ODEs with 28 parameters and 15 variables, which is shown in Figure 5.2.

Out of the equations, the second equation for the MPF concentration is the most relevant and the GUI plots its simulation results.

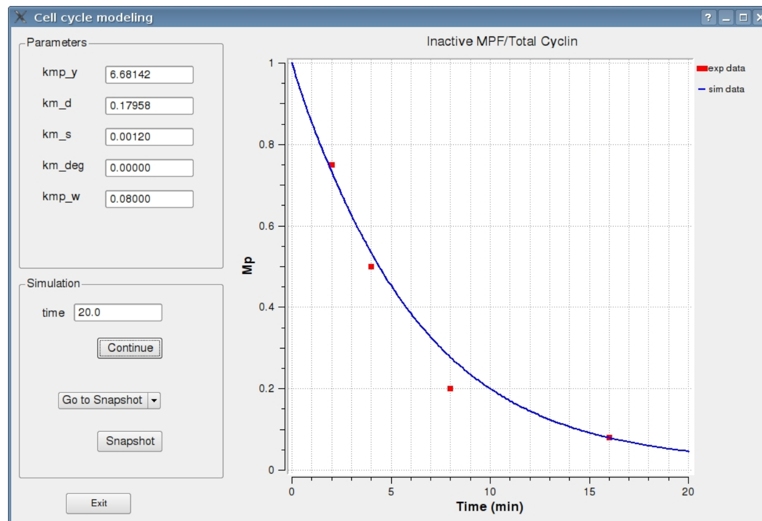
$$\frac{dMp}{dt} = -(km_d \times (Dp + Dsp) \times Mp) + (kmp_y \times Myt + kmp_w \times Wp) \times M - (km_deg \times Mp)$$

Figure 5.3a shows the GUI with the initial configuration for simulating the cell cycle of frog-egg extracts. The simulated MPF concentration is plotted in the right. The 4 red squares in the plotting area represents the actual MPF concentration data obtained in "wet" lab experiments for reference: 0.75 at 2 minutes, 0.5 at 2, 0.2 at 8, and 0.08 at 16.

The upper left `Parameters` pane shows 5 parameters and corresponding input area where the user can change each parameter values during the simulation. The parameter values are initially set using empirically determined "good" values for this specific simulation (i.e., dephosphorylation of pre-MPF during mitosis). The lower left `Simulation` pane embeds the user controls for the simulation. The user can specify the simulation time in the `time` input area and execute the simulation using the specified parameter values by clicking the `Run` button. Also, the user can save current simulation state by clicking the `Snapshot` button and restore saved states through the `Go to Snapshot` box.



(a) GUI layout and initial simulation configuration



(b) Simulation with no adaptation

Figure 5.3: Cell cycle simulation of frog-egg extracts

As an example usage of the GUI, Figure 5.3b plots simulation results until 20 minutes. The simulation uses the initial parameter values, which seem to perform quite effectively because the simulated Mp values close match the real experimental data. We note that this simulation does not involve any adaptations in simulating the model from time 0 to time 20, which is the usual way most simulation tools behaves like PET.

5.4.1 Dynamic Parameter Change

Figure 5.4 shows an example adaptive simulation where the model parameter values are dynamically changed by the user. First, the user specifies how long he or she simulate the model, what parameter values are to be used, and starts the simulation. Figure 5.4a plots the simulation results where the simulation end time is set to 2 minutes and the initial

parameter values are used. Then, after observing and analysing the simulation results, the user decides to change two parameters from there default values – *kmp_y* to 0.5 and *km_d* to 1.0. Wanting to know how the simulated system progresses if the new setting is kept for 2 minutes, the user sets the simulation time to 4 minutes and continues the simulation by clicking the `Continue` button (the text of this button has changed from `Run` to `Continue` in the previous 2-minute simulation). The MPF plot in Figure 5.4b shows how the simulated system evolves by the newly set parameter values, where the MPF concentration drops more rapidly than its previous setting. Finally, after checking the simulation results, the user makes another parameter change decision to change *km_d* to 0.5 and simulates the system until 20 minutes. The plot in Figure 5.4c shows how the MPF concentration is changed by the new setting of the simulation, where its decrease rate becomes slow after time 4.

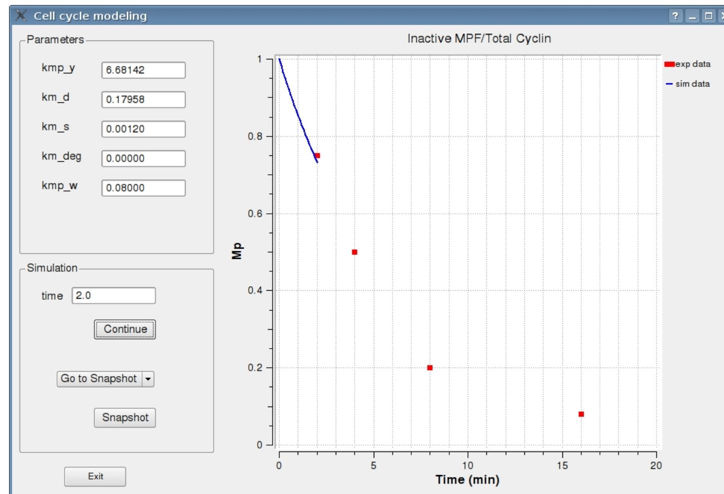
As demonstrated by this example, the dynamic adaptation capability implemented in this work allows the user to explore new pathways in evolving biological systems in such a flexible way that does not require stop and restart of an on-going simulation.

5.4.2 Simulation Rollback and Continuation

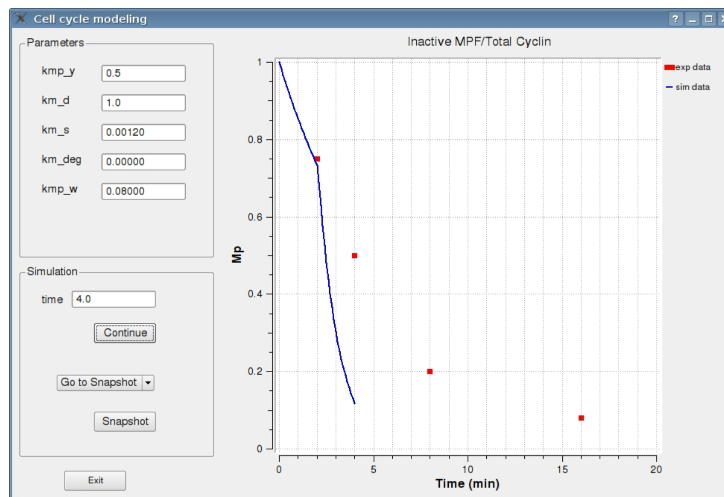
Figure 5.5 shows an example of how a simulation can rollback and continue by our dynamic adaptation method. First, by clicking the `Snapshot` button, the user saves a simulation state which the user thinks is interesting to reuse in future experiments. A saved state contains entire information needed to resume a simulation starting from the saved point, including all the model parameter values and variable values for the duration of simulated evolution. Later, saved states can be restored by choosing a specific state in the `Go to Snapshot` box. For example, Figure 5.5a shows a restored state (`snapshot 1`) from Figure 5.4a in the previous parameter change adaptation example. Using the saved results as a new starting point, the user then tweaks two parameters, *kmp_y* and *km_d*, by setting their values to 0.5 and 0.5, respectively, and continues the simulation until 10 minutes. The simulation results are shown in Figure 5.5b, where the MPF concentration decreases more rapidly by the changes in the reaction rates. At this point, the user can explore new pathways in evolving the system. As shown in Figure 5.5c, the user changes *kmp_y* and *km_d* again to 1.0 and 1.0, respectively, and continues the simulation until time 20. Affected by this reaction rate change, the MPF plot in Figure 5.5c shows more rapid decrease in its concentration after time 10.

5.5 Related Work

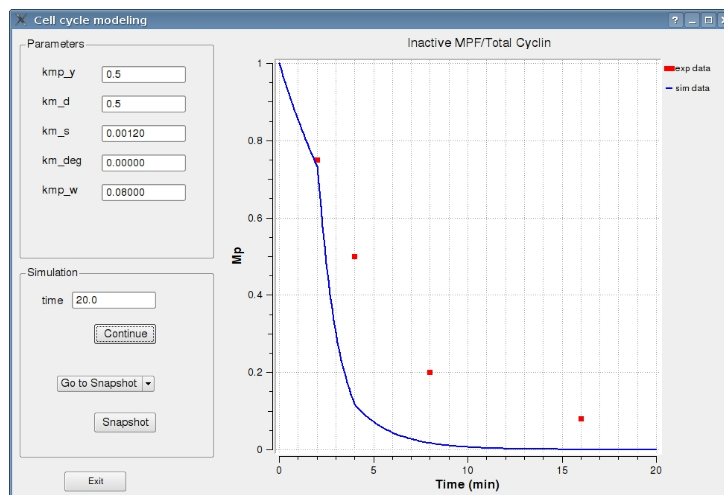
Our method is similar to computational steering (see Section 2.7), whereby computation can be dynamically adapted through an interactive control to change parameter values of a simulated system. While implementing computational steering requires tight integration of the steering environment and target applications, our approach supports modular development of adaptation code modules and transparent integration with a give application, thus composing an adaptive application. In addition, our approach provides more flexibility in performing scientific simulations by allowing the user to save intermediate simulation results on-the-fly and restore them for later use.



(a) Simulation until time 2 with initial parameter values

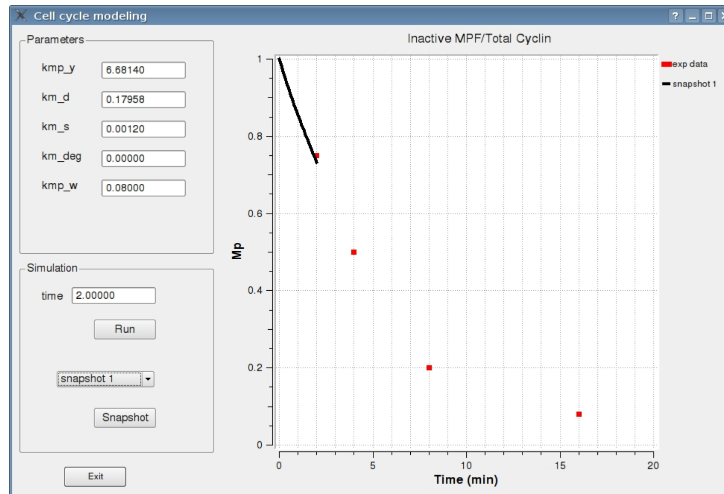


(b) Simulation until time 4 with *kmp_y=0.5* and *km_d=1.0*

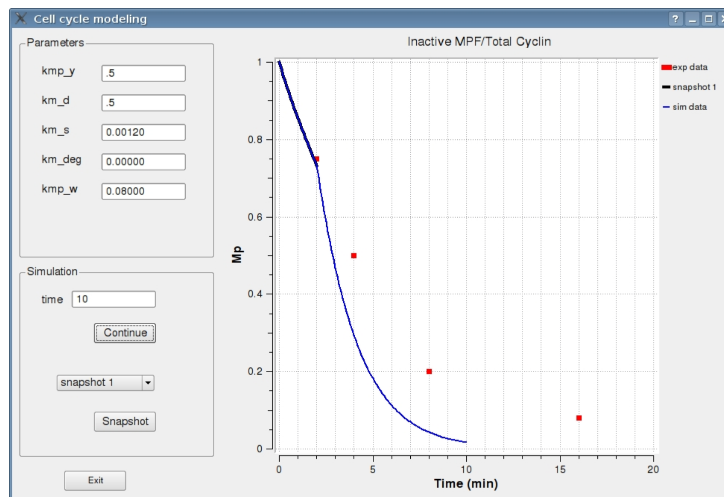


(c) Simulation until time 20 with *kmp_y=0.5* and *km_d=0.5*

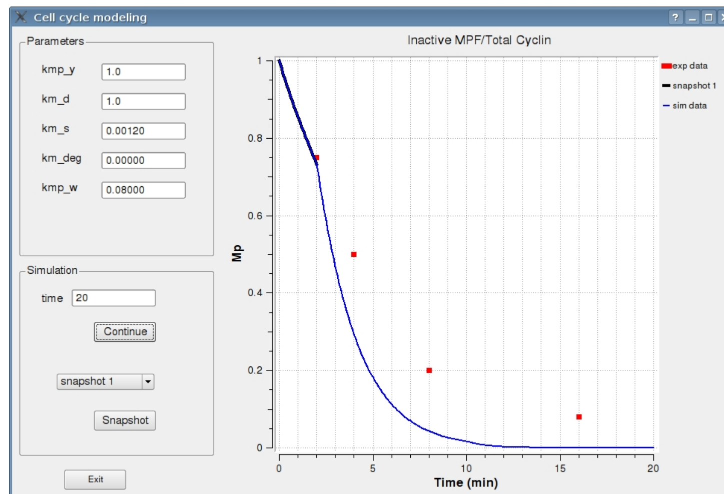
Figure 5.4: Dynamic parameter change in cell cycle simulations



(a) Restored state at time 2 (Figure 5.4a) through rollback



(b) Simulation until time 10 with *kmp_y=0.5* and *km_d=0.5* from the restored state



(c) Simulation until time 20 with *kmp_y=1.0* and *km_d=1.0*

Figure 5.5: Runtime simulation rollback and continuation in cell cycle simulations

5.6 Summary

In this study, we presented a modular approach to implementing dynamic adaptations to support loosely defined adaptation schemes. Our approach allows domain-specific adaptation schemes to be implemented in a separate code and then combined with a given scientific program to compose an application enhanced with adaptive behavior. An adaptive application implemented through our approach provides a great amount of flexibility in performing simulations by realizing various user-driven adaptation scenarios at runtime. We demonstrated the benefits of this increased flexibility through applications to cell cycle modeling and simulations.

Chapter 6

Conclusions

This thesis has presented new approaches to adapting scientific software in a modular way. In the design level, our aspect-oriented implementations of adaptation patterns provide code templates that can be reused by extending subclasses. In the implementation level, we presented a compositional approach to implementing adaptations in a modular way, so that the programmer can focus on the design of application-specific adaptation strategies. We demonstrated the effectiveness of our approaches by implementing various adaptations strategies in the scientific computing area.

Facing the unprecedented challenges of modern scientific applications requires the adoption of novel software development techniques and approaches. In that light, the modularity advantages offered by our approaches can offer viable solutions to these challenges. The ideas presented in this thesis can help address the challenges of emerging scientific software.

6.1 Thesis Contributions

Specific contributions of this thesis are summarized as follows:

- In Chapters 3 and 5, we presented a modular and language-independent approach to realizing adaptation schemes for scientific applications. Using a compositional framework based on function call interception and manipulation, the adaptive logic to monitor internal program states and control the behavior of program modules is written and managed as a separate code, thus supporting centralized design of complex adaptation strategies for adapting to dynamic changes within an application.

- In Chapter 3, we presented a dynamic method for tuning algorithmic parameters of parallel scientific programs. By treating tuning as a separate concern in the software development process, our method supports application-specific development of optimization schemes for existing programs that are not easily supported by conventional tuning techniques. Our dynamic tuning approach accounts for runtime factors such as input problem size and parallel characteristics of a given execution platform, as well as the architectural or runtime properties of a single machine of the platform.

- In Chapter 5, we presented a modular approach to implementing loosely defined adaptations of scientific applications. Through our approach, an adaptation code that

implements loosely defined schemes can control the execution of an application, so that the user can make adaptive decisions based on observed results at runtime and dynamically change the state of a running computation. An adaptive application implemented in this way provides greater amount of flexibility in performing simulations than conventional applications by realizing various user-driven adaptation scenarios at runtime.

- In Chapter 4, we presented a novel approach to adapting scientific software written in Fortran. Our approach expresses the adaptability functionality as `abstract` aspects that implement known adaptation patterns in scientific computing. The aspect-oriented pattern implementations are reusable and maintainable across multiple scientific applications, thereby increasing programmer productivity by removing duplication and leveraging aspect inheritance.

6.2 Future Work

Potential research directions that extend the work in this thesis are as follows:

- **Dynamic Tuning:** To further examine the applicability of the presented dynamic tuning approach, we plan to apply our approach to parallel applications in other domains and investigate possible issues.
- **Adaptivity schema library:** Future work directions will focus on providing a more complete library of adaptivity schema aspects to support additional adaptivity schemas such as staged composition and problem decomposition [24].
- **Expressibility of AOP-based adaptation:** Our AOP-based approach to adapting scientific applications currently supports only the `execution` pointcut. Therefore, it is natural to investigate how to support additional AOP constructs to increase the expressive power of our approach.

Bibliography

[1] Vladmir Estivill-Castro and Derick Wood. A Survey of Adaptive Sorting Algorithms. *ACM Comput. Surv.*, 24(4):441–476, 1992. ISSN 0360-0300.

[2] Richard Barrett, Michael Berry, Jack Dongarra, Victor Eijkhout, and Charles Romine. Algorithmic Bombardment for the Iterative Solution of Linear Systems: A Poly-iterative Approach. *J. Comput. Appl. Math.*, 74(1-2):91–109, 1996. ISSN 0377-0427.

[3] Sanjukta Bhowmick, Lois C. McInnes, Boyana Norris, and Padma Raghavan. The Role of Multi-method Linear Solvers in PDE-based Simulations. In V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *Computational Science and Its Applications - ICCSA 2003, Part I*, volume 2667, pages 828–839. Springer, 2003. ISBN 3-540-40155-5.

[4] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. In Henry G. Dietz, editor, *LCPC*, volume 2624 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2001. ISBN 3-540-04029-3.

[5] Hao Yu, Dongmin Zhang, and Lawrence Rauchwerger. An Adaptive Algorithm Selection Framework. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 278–289, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2229-7.

[6] Michael J. Voss and Rudolf Eigemann. High-level Adaptive Program Optimization with ADAPT. In *PPoPP '01: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 93–102, New York, NY, USA, 2001. ACM. ISBN 1-58113-346-4.

[7] Brian Ensink, Joel Stanley, and Vikram Adve. Program Control Language: A Programming Language for Adaptive Distributed Applications. *J. Parallel Distrib. Comput.*, 63 (11):1082–1104, 2003. ISSN 0743-7315.

[8] Wei Du and Gagan Agrawal. Language and Compiler Support for Adaptive Applications. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 29, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2153-3.

[9] Ken Kennedy, Mark Mazina, John M. Mellor-Crummey, Keith D. Cooper, Linda Torczon, Francine Berman, Andrew A. Chien, Holly Dail, Otto Sievert, Dave Angulo,

Ian T. Foster, Ruth A. Aydt, Daniel A. Reed, Dennis Gannon, S. Lennart Johnsson, Carl Kesselman, Jack Dongarra, Sathish S. Vadhiyar, and Richard Wolski. Toward a Framework for Preparing and Executing Adaptive Grid Programs. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 322, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1573-8.

[10] Det Buaklee, Gregory F. Tracy, Mary K. Vernon, and Stephen J. Wright. Near-Optimal Adaptive Control of a Large Grid Application. In *ICS '02: Proceedings of the 16th International Conference on Supercomputing*, pages 315–326, New York, NY, USA, 2002. ACM. ISBN 1-58113-483-5.

[11] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid using AppLeS. *Parallel and Distributed Systems, IEEE Transactions on*, 14(4):369–382, April 2003. ISSN 1045-9219.

[12] Vladimir Janjic, Kevin Hammond, and Yang Yang. Using Application Information to Drive Adaptive Grid Middleware Scheduling Decisions. In *MAI '08: Proceedings of the 2nd Workshop on Middleware-Application Interaction*, pages 7–12, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-204-7.

[13] Netlib. http://www.netlib.org.

[14] Paul F. Dubois. Ten Good Practices in Scientific Programming. *Computing in Science and Engg.*, 1(1):7–11, 1999. ISSN 1521-9615.

[15] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[16] Viktor K. Decyk, Charles D. Norton, and Henry J. Gardner. Why Fortran? *Computing in Science and Engineering*, 9(4):68–71, 2007. ISSN 1521-9615.

[17] Erik Hilsdale and Jim Hugunin. Advice Weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-842-3.

[18] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 196–205, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-662-X.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-056-6.

[20] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[21] AspeCt-oriented C Compiler. http://www.aspectc.net.

[22] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *CRPIT '02: Proceedings of the 40th International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, 2002. Australian Computer Society, Inc. ISBN 0-909925-88-7.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4.

[24] Srinidhi Varadarajan and Naren Ramakrishnan. Novel Runtime Systems Support for Adaptive Compositional Modeling in PSEs. *Future Gener. Comput. Syst.*, 21(6):878–895, 2005. ISSN 0167-739X.

[25] Gabrielle Allen, Thomas Dramlitsch, Ian Foster, Nicholas T. Karonis, Matei Ripeanu, Edward Seidel, and Brian Toonen. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 52–52, New York, NY, USA, 2001. ACM. ISBN 1-58113-293-X.

[26] Fangzhe Chang and Vijay Karamcheti. A Framework for Automatic Adaptation of Tunable Distributed Applications. *Cluster Computing*, 4(1):49–62, 2001. ISSN 1386-7857.

[27] Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny. Adaptive Scheduling for Master-Worker Applications on the Computational Grid. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 214–227, London, UK, 2000. Springer-Verlag. ISBN 3-540-41403-7.

[28] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, page 55, Washington, DC, USA, 2001. IEEE Computer Society.

[29] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor-A Hunter of Idle Workstations. *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, Jun 1988.

[30] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1997.

[31] Alan Su, Francine Berman, Richard Wolski, and Michelle Mills Strout. Using Apples to Schedule Simple SARA on the Computational Grid. *Int. J. High Perform. Comput. Appl.*, 13(3):253–262, 1999. ISSN 1094-3420.

[32] Francine D. Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (CDROM)*, page 39, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-89791-854-1.

[33] Neil Spring and Rich Wolski. Application Level Scheduling of Gene Sequence Comparison on Metacomputers. In *ICS '98: Proceedings of the 12th International Conference on Supercomputing*, pages 141–148, New York, NY, USA, 1998. ACM. ISBN 0-89791-998-X.

[34] Daniel S. Myers and Adam L. Bazinet. Intercepting Arbitrary Functions on Windows, UNIX, and Macintosh OS X Platforms. Technical Report CS-TR-4585, UMIACS-TR-2004-28, Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies, University of Maryland, 2004.

[35] Yannis Smaragdakis. Layered Development with (Unix) Dynamic Libraries. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 33–45, London, UK, 2002. Springer-Verlag. ISBN 3-540-43483-6.

[36] John R. Levine. *Linkers and Loaders*, chapter 8. Morgan Kaufmann, San Francisco, CA, 2000.

[37] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–144, July 1999.

[38] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. http://java.sun.com/docs/books/vmspec/.

[39] Michael Alan Heffner. A Runtime Framework for Adaptive Compositional Modeling. Master's thesis, Blacksburg, VA, USA, 2004.

[40] Pilsung Kang, Michael A. Heffner, Naren Ramakrishnan, Calvin J. Ribbens, and Srinidhi Varadarajan. Adaptive Code Collage: A Framework to Transparently Modify Scientific Codes. *IEEE Computing in Science and Engineering (under review)*, under review.

[41] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, September 2004.

[42] Alexandre Ern, Vincent Giovangigli, David E. Keyes, and Mitchell D. Smooke. Towards Polyalgorithmic Linear System Solvers for Nonlinear Elliptic Problems. *SIAM J. Sci. Comput.*, 15(3):681–703, 1994. ISSN 1064-8275.

[43] Sanjukta Bhowmick, Lois McInnes, Boyana Norris, and Padma Raghavan. Robust algorithms and software for parallel PDE-based simulations. In *Proceedings of the Advanced Simulation Technologies Conference, ASTC'04, April 18-22, 2004*. Society for Modeling and Simulation International (SCS), 2004.

[44] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–288, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9.

[45] Troy A. Johnson and Rudolf Eigenmann. Context-Sensitive Domain-Independent Algorithm Composition and Selection. *SIGPLAN Not.*, 41(6):181–192, 2006. ISSN 0362-1340.

[46] Robert Marshall, Jill Kempf, Scott Dyer, and Chieh-Cheng Yen. Visualization Methods and Simulation Steering for a 3D Turbulence Model of Lake Erie. *SIGGRAPH Comput. Graph.*, 24(2):89–97, 1990. ISSN 0097-8930.

[47] Steven G. Parker and Christopher R. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*, page 52, New York, NY, USA, 1995. ACM. ISBN 0-89791-816-9.

[48] Jurriaan D. Mulder, Jarke J. van Wijk, and Robert van Liere. A Survey of Computational Steering Environments. *Future Gener. Comput. Syst.*, 15(1):119–129, 1999. ISSN 0167-739X.

[49] H. Wright, R.H. Crompton, S. Kharche, and P. Wenisch. Steering and Visualization: Enabling Technologies for Computational Science. *Future Generation Computer Systems*, 2008.

[50] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3.

[51] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9.

[52] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27:3–35, 2001.

[53] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *Proc. of SciDAC 2005, Journal of Physics: Conference Series*, volume 16, pages 521–530. Institute of Physics Publishing, June 2005.

[54] Danesh Tafti. GenIDLEST - A Scalable Parallel Computational Tool for Simulating Complex Turbulent Flows. In *Proceedings of the ASME Fluids Engineering Division (FED)*, volume 256. ASME-IMECE, November 2001.

[55] Pilsung Kang, Naresh K. C. Selvarasu, Naren Ramakrishnan, Calvin J. Ribbens, Danesh K. Tafti, and Srinidhi Varadarajan. Modular, Fine-Grained Adaptation of Parallel Programs. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 269–279. Springer, May 2009.

[56] K. Seymour, Haihang You, and J. Dongarra. A Comparison of Search Heuristics for Empirical Code Optimization. In *iWAPT 2008: the 3rd International Workshop on Automatic Performance Tuning*, pages 421–429, 29 2008-Oct. 1 2008.

[57] Jack Dongarra and Robert Schreiber. Automatic Blocking of Nested Loops. Technical report, Knoxville, TN, USA, 1990.

[58] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[59] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A Comparison of Empirical and Model-driven Optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 63–76, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5.

[60] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing Matrix Multiply using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM. ISBN 0-89791-902-5.

[61] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R.C. Whaley, and K. Yelick. Self-Adapting Linear Algebra Algorithms and Software. *Proceedings of the IEEE*, 93(2):293–312, Feb. 2005. ISSN 0018-9219.

[62] Dong Kwan Kim, Yang Jiao, and Eli Tilevich. Flexible and Efficient In-Vivo Enhancement for Grid Applications. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 444–451, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3622-4.

[63] Paul D. Hovland and Michael T. Heath. Adaptive SOR: A Case Study in Automatic Differentiation of Algorithm Parameters. Technical Report ANL/MCS-P673-0797, Mathematics and Computer Science Division, Argonne National Laboratory, 1997.

[64] K. Radhakrishnan and A. C. Hindmarsh. Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations. Technical Report UCRL-ID-113855, LLNL, 1993.

[65] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26(4):345–420, 1994. ISSN 0360-0300.

[66] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[67] F2PY: Fortran to Python interface generator. http://cens.ioc.ee/projects/f2py2e/.

[68] Pilsung Kang, Naresh K. C. Selvarasu, Naren Ramakrishnan, Calvin J. Ribbens, Danesh K. Tafti, and Srinidhi Varadarajan. Dynamic Tuning of Algorithmic Parameters of Parallel Scientific Codes. In *ICCS '10: Proceedings of the 10th International Conference on Computational Science*, pages 145–153, May 2010.

[69] Thomas J. McCabe. A Complexity Measure. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[70] Lawrence Livermore National Laboratory. http://computation.llnl.gov/casc/components/babel.html.

[71] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 13, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0287-3.

[72] John Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-Oriented Programming of Sparse Matrix Code. In *ISCOPE '97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, pages 249–256, London, UK, 1997. Springer-Verlag. ISBN 3-540-63827-X.

[73] Brian Amedro, Vladimir Bodnartchouk, Denis Caromel, Christian Delbe, Fabrice Huet, and Guillermo Taboada. Current State of Java for HPC. Technical Report RT-0353, INRIA, 2008.

[74] Bruno Harbulot and John R. Gurd. Using AspectJ to Separate Concerns in Parallel Scientific Java Code. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 122–131, New York, NY, USA, 2004. ACM. ISBN 1-58113-842-3.

[75] Hyuck Han, Hyungsoo Jung, Heon Y. Yeom, and Dong-Young Lee. Taste of AOP: Blending Concerns in Cluster Computing Software. In *CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 110–117, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1387-4.

[76] J.L. Sobral. Incrementally Developing Parallel Applications with AspectJ. *Parallel and Distributed Processing Symposium, International*, 0:95, 2006.

[77] Bruno Harbulot and John R. Gurd. A Join Point for Loops in AspectJ. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 63–74, NY, USA, 2006. ACM. ISBN 1-59593-300-X.

[78] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.0, May 2008. http://www.openmp.org.

[79] JBOSS AOP. http://www.jboss.org/jbossaop.

[80] AspectJ. http://www.eclipse.org/aspectj.

[81] Carlos A. Cunha, João L. Sobral, and Miguel P. Monteiro. Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms. In *AOSD '06: Proceedings of the 5th International conference on Aspect-Oriented Software Development*, pages 134–145, New York, NY, USA, 2006. ACM. ISBN 1-59593-300-X.

[82] PET (Parameter Estimation Toolkit). http://mpf.biol.vt.edu/pet.

[83] A. C. Hindmarsh. ODEPACK, A Systematized Collection of ODE Solvers , R. S. Stepleman et al. (eds.), North-Holland, Amsterdam, (vol. 1 of), pp. 55-64. *IMACS Transactions on Scientific Computation*, 1:55–64, 1983.

[84] Nokia. http://qt.nokia.com.

[85] IEEE and The Open Group. IEEE Standard 1003.1-2001, 2001.

[86] Jason W. Zwolak, John J. Tyson, and Layne T. Watson. Parameter Estimation for a Mathematical Model of the Cell Cycle in Frog Eggs. *Journal of Computational Biology*, 12(1):48–63, 2005.

[87] A Kumagai and W G Dunphy. Control of the Cdc2/cyclin B Complex in Xenopus Egg Extracts Arrested at a G2/M Checkpoint with DNA Synthesis Inhibitors. *Mol. Biol. Cell*, 6(2):199–213, 1995.

[88] Ulrich Drepper. How to Write Shared Libraries. http://people.redhat.com/drepper/dsohowto.pdf version 4.0.

[89] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, San Francisco, CA, 2000.

[90] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000. ISSN 1094-3420.

[91] Alexandre Vasseur. Dynamic AOP and Runtime Weaving for Java - How does AspectWerkz Address It? In R. E. Filman, M. Haupt, K. Mehner, , and M. Mezini, editors, *Proceedings of the 2004 Dynamic Aspect Workshop (DAW'04)*, pages 135–145, March 2004.

[92] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *AOSD '02: Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 141–147, New York, NY, USA, 2002. ACM. ISBN 1-58113-469-X.

[93] Dong Kwan Kim and Eli Tilevich. Overcoming JVM HotSwap Constraints via Binary Rewriting. In *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, pages 1–5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-304-4.

Appendix A

Program Compilation and Linking

A program written in a high-level language goes through multiple transformation steps until it gets loaded in memory as a process and executed by the processor. The transformation steps are described here.

A.1 Stages in Program Build Process

In general, the compilation or build process to generate an executable program image from source code written in high-level languages such as C consists of a set of multiple, cascaded stages. Figure A.1 shows these steps when an application is built through collaboration of a compiler and a linker (assuming C code). Each stage is described in the following:

1. **Preprocessing**: Header files are included (copied to the source code) and macros are expanded. Typically, giving the "-E" option to the compiler (e.g., "gcc -E A.c") makes the compiler stop after the preprocessing step.

2. **Compilation Proper**: Preprocessed code is translated to assembly code. The "-S" option makes the compiler generate the assembly code and stop. Let us call the time this process occurs, *compilation proper time*.

3. **Assembly**: The assembly code generated from the compilation proper step is assembled to machine code or native code. However, the generated file, which is commonly called an *object file* and whose filename extension is typically .o, is not yet directly executable. First, there can be unresolved references (symbols), which need to be replaced with the actual address at which corresponding data or code (definitions) will be loaded. For instance, if the original code used *printf()*, a C standard function whose implementation is provided by the C runtime system of the execution platform, its reference in the generated object file is not yet resolved with a corresponding definition (implementation). Binding symbols with definitions, which is called *symbol resolution*, is performed by the linker in the linking phase.

   Second, the actual load addresses for the data and the code in an object file are not yet determined. Since the base address, the reference point for other addresses within an object file, is zero in each object file, memory addresses need to be recalculated when

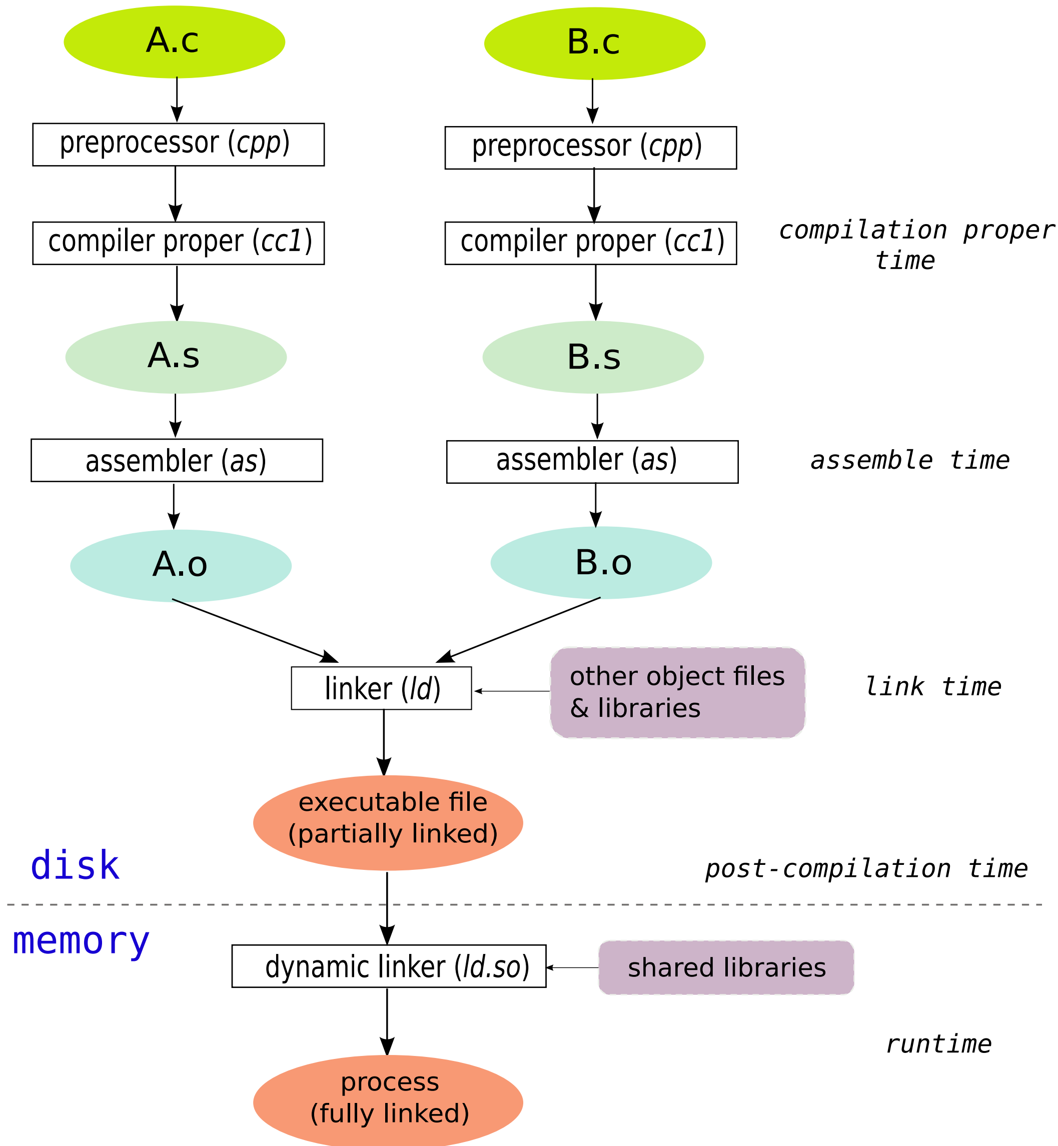


Figure A.1: Compilation and Linking Steps Involved to Create a Process from Source Files

multiple object files (and the library code that is used by the source code) are combined to form an executable program image. In other words, data and code in object files need to be *relocated* when combined to fix their actual load addresses. Since object files need relocation, they are also called *relocatable objects*.

The "-c" option makes the compiler generate the object file and stop. The time the assembly step occurs and an object file is generated is commonly called *assemble time* or *assembly time*.

4. **Linking**: One or more relocatable objects are combined, into an executable program. As described above, data and code in separate object files need to combined through symbol resolution and relocation process by a linker program . Library codes, if used, take part in the linking process too, but linking with libraries needs further explana-

tion because there are two forms of libraries, static and dynamic, which are treated differently by the linker.

A static library, whose filename extension is typically .a, is just a collection of one or more relocatable object files. Therefore, static libraries are treated in the same manner as with object files; code and data are copied to an executable through symbol resolution and relocation. Linking with static libraries and/or relocatable objects is called *static linking* and the linker that performs static linking, such as *ld* in Unix systems, is called a static linker.

A dynamic library, whose filename extension is commonly .so, is a compiled code to be loaded at application launch time or at runtime. While static library code is copied to every program image that uses the code, dynamic library code is not directly copied to an executable in the linking process. Instead, the information about dynamic libraries used by a program is stored in an executable, such that those libraries are correctly located and loaded at runtime. References to dynamic libraries remain unresolved, making the generated executable file partially linked.

The code of a dynamic library is typically compiled *position-independent*, usually with the "-fPIC" compiler option, so that any location in the address space of a process can be assigned to the code. On modern operating systems, this position independence allows one dynamic library code to be shared among multiple processes and to efficiently execute at a different address in each process. Dynamic libraries are also known as dynamically linked libraries, shared libraries, shared objects, or dynamically shared objects (DSOs). Linking with dynamic libraries is called *dynamic linking* and the program that performs dynamic linking is called a dynamic linker, which is *ld-linux.so* on Linux and *ld.so* on BSD systems.

Position-Independent Code (PIC): machine code that can be accessed and executed regardless of where it is loaded in memory. Shared libraries – dynamic shared objects – in modern systems use PIC, so that one copy of the code is mapped to multiple address spaces of different processes at the same time. Modern Unix systems typically implement PIC through indirection tables, which are generated by the linker at build time, such as Global Offset Table (GOT) and Procedure Linkage Table (PLT), so that both data and code can be accessed independently of actual load address.

The time static linking happens and an executable file is generated is most commonly called *compile time* or *compilation time*. However, for discussion of the details of an application build process here, let us call it *static link time* or *link time* in short.

The *post-compilation time* is an in-between time before an executable file is launched to execute after linking is finished to generate the executable.

A.2 Dynamic Linking

Dynamic libraries are loaded and linked by a dynamic linker at application launch time or at runtime. Shared library code is mapped to the address space of an application process and unresolved symbols to data and code in a shared library are automatically resolved by the dynamic linker, thereby making the launched program fully linked. In this process of dynamic linking, any dynamic libraries used by an application need to be specified (together with location information) when building an executable file.

*Dlopen*ing is another type of dynamic linking, where an application can explicitly control loading of a shared library through the API functions to the dynamic linker such as *dlopen()* in Unix systems and *LoadLibrary()* in Windows systems. Symbols defined in a dlopened library can be accessed through the *dlsym()* function in Unix and *GetProcAddress()* in Windows, so that they can be used by the application that loaded the dynamic module. Information about the modules to be dlopened need not be specified when building an executable file, meaning the modules need not be available at build time. The location information is needed only at runtime when dlopening. Dlopening is also often called *dynamic loading* and the library modules loaded by dlopening are sometimes referred to as *dynamically loaded modules*.

Technical details on dynamic linking and shared libraries can be found in [88]. Linkers and loaders surveyed in [89].

A.3 Code Modification Tools

A.3.1 System dynamic linker

By using the dlopen API offered by a dynamic linker, an application can be programmed by a user to explicitly load a DSO dynamically, so that the code in dlopened modules can be accessed and used in the application. Dlopening can be used to implement adaptive scenarios, such as replacing a function with another function in the newly loaded module, a basic operation for implementing runtime algorithm switching.

A.3.2 LLL

LLL's dynamic linking and loading functionality can be used for similar adaptation purposes as with dlopening, such as runtime algorithm switching. However, LLL allows an application to execute even when there are unresolved symbols in a DSO, whereas system dynamic linkers require every symbol to be resolved. The web browser example with LLL shows this capability, where the browser can keep functioning with only critical component modules are loaded, while insignificant modules, such as for image rendering, are not loaded. Although a bunch of symbols to unloaded modules remain unresolved, which would cause runtime errors and stop the application if loaded with a dynamic linker, the browser can keep running and perform tasks that can be handled with loaded modules.

A.3.3 Callee-site modification tools

Detours [37] and DynInst [90] transform the code at the callee-site of a function call, such as by replacing the first few instructions of an intercepted function with a jump to a user-provided function that handles the intercepted call (as in handler functions in ACC). These tools use binary rewriting of in-memory code, thus modifying the original code when the application is in execution.

Detours

Detours is a library for instrumenting arbitrary Win32 binary functions (32-bit API for Windows systems), so that th calls to target functions can be intercepted and detoured to the user-provided *detour* function, which can replace or extend the target function. For example, if a programmer is writing a program that uses a Win32 function and want to intercept it, he can include the Detours library in the program and use its API to intercept the Win32 functions.

For target functions that are not available for linking, such as the functions in a dynamically loaded library (through *LoadLibrary*), there are several methods to access the target function; Often a pointer to the target function can be acquired in another function (through *GetProcAddress*, which is equivalent to *dlsym()* in Unix), or the Detours library offers a *DetourFindFunction()* to find the pointer to a target function.

However, one weak point of Detours is that, since Detours only supports instrumenting Win32 functions, it is not applicable for intercepting non-Win32 functions, such as user-written functions, thus limiting its uses in general.

DynInst

DynInst offers an API for inserting code to a running program on both Unix and Windows systems. Code modification places are usually the entry or exit (or both) of a function of interest. To modify a program in execution (referred to as *mutatee*), a separate program (*mutator*), where the programmer uses the DynInst API to instrument mutatee and perform mutating operations, is executed with the information of mutatee (e.g., the process id of the mutatee). Functions in dynamically loaded modules in an application can be effectively handled since DynInst allows code modification at any time during program execution. A mutator program, which can use the DynInst API to find the target function to instrument with the function name, can be executed once a DSO is loaded and its symbols are available to the mutatee process.

A.3.4 Java tools

Java AOP tools insert *advice code*, a piece of code that can change runtime behavior of an existing program at control points, transparently to the original code through *weaving*. Weaving can be performed at compile time [17], post-compilation time [17], class load time [80, 91], or at runtime [92].

The HotSwap functionality available since Java 1.4 enables replacing classes dynamically, which is useful for implementing diverse adaptive scenarios such as runtime algorithm

switching. Limitations of HotSwap, such as requiring the same class signature in replacing classes, can be overcome through bytecode rewriting [93].

A.4 Executing ACC Initialization Code before C *main()*

The ACC initialization operations, such as setting up a parameter list for target functions to intercept or registering handlers to associate with intercepted functions, need to be performed early before the target functions are called. If the glue code, which combines the original source code and newly developed adaptation code, can define C *main()* inside, it would be an ideal to put the initialization operations since they can be placed there to execute early in program execution. This is the case, as described in the paper, for programs developed for use as a library and Fortran applications whose symbol name of the entry point, when exported for linking, does not interfere with C *main()*.

For C applications with a *main()*, the glue code cannot define another *main()* because it conflicts with that of the original code. In such cases, there are two possible solutions to avoiding direct modification of the original code to include the ACC initialization operations.

1. Intercept a function called by *main()*: After all, the original source code needs to be available to apply ACC. We can look at the *main()* function in the original program and choose a function that is called early (e.g., some initialization function executed in the program) to intercept, so that a handler code can be written to execute the ACC initialization operations when the function is intercepted to transfer the execution control to the handler.

2. Use the *constructor* function attribute: Although not a C standard, the *constructor* function attribute is supported as an extension by many C compilers, such as GCC, Intel, and IBM compilers, enabling a function with the attribute to execute before *main()*. By defining a constructor function and placing any necessary ACC initializations in it, the *main()* function in the original code need not be directly touched. Listing A.1 shows a sample constructor function for the sorting algorithm switching example.

```
void __attribute__ ((constructor)) init(void)
{
    acc_init("sortswitching");

    ACC_PARAM_TYPE params[3];
    struct acc_symbol_entry* se;

    /* create and add symbol "quicksort" for qsort */
    se = acc_add_symbol("quicksort", (acc_invoke_func_t *)quicksort, 1);

    /* assign type for each parameter */
    params[0] = ACC_PARAM_POINTER;
    params[1] = ACC_PARAM_INT;
    params[2] = ACC_PARAM_INT;
```

```
    /* associate the parameter list with qsort symbol entry,
       specifying the size of the list  */
    acc_add_params_list(se, params,3);

    /* register a pre-handler for qsort */
    acc_add_handler(ACC_HT_PRE, "quicksort", qsorthandler);

    return;
}
```

Listing A.1: Constructor function with ACC initializations for the sorting algorithm switching example