Maintainable and Reusable Scientific Software Adaptation

Democratizing Scientific Software Adaptation

Pilsung Kang
Flash Solution Software Development
Samsung Electronics, Korea
pils.kang@samsung.com

Eli Tilevich, Srinidhi Varadarajan, and Naren Ramakrishnan
Center for High-End Computing Systems
Dept. of Computer Science, Virginia Tech
Blacksburg, VA 24061, USA
{tilevich,srinidhi,naren}@cs.vt.edu

ABSTRACT

Scientific software must be adapted for different execution environments, problem sets, and available resources to ensure its efficiency and reliability. Although adaptation patterns can be found in a sizable percentage of recent scientific applications, the traditional scientific software stack lacks the adequate adaptation abstractions and tools. As a result, scientific programmers manually implement ad-hoc solutions that are hard to maintain and reuse. In this paper, we present a novel approach to adapting scientific software written in Fortran. Our approach leverages the binary object code compatibility between stack-based imperative programming languages. This compatibility makes it possible to apply a C++ Aspect-Oriented Programming (AOP) extension to Fortran programs. Our approach expresses the adaptive functionality as `abstract` aspects that implement known adaptation patterns and can be reused across multiple scientific applications. Application-specific code is systematically expressed through inheritance. The resulting adaptive functionality can be maintained by any programmer familiar with AOP, which has become a staple of modern software development. We validated the expressive power of our approach by refactoring the hand-coded adaptive functionality of a real-world computational fluid dynamics application suite. The refactored code expresses the adaptive functionality in 27% fewer ULOC on average by removing duplication and leveraging aspect inheritance.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.13 [**Reusable Software**]: [Reusable libraries]; D.3.3 [**Language Constructs and Features**]: [Frameworks, Patterns]

General Terms

Languages, Management

Keywords

scientific software, program adaptation, aspect-oriented programming, software maintenance

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 known 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 [20], 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 functionality can also be executed dynamically to adapt an application's semantics based on some runtime conditions.

Several researchers have recently identified dynamic adaptation as capable of benefiting scientific software [9, 21]. Our own experiences of collaborating with scientific programmers confirm that the traditional execution model of scientific applications—build, run, change, run anew—no longer provides the 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.

Based on our ongoing collaboration with scientific programmers, we have observed a trend in which esoteric solutions to dynamic adaptation are crafted for individual applications, leading to adaptations that are neither maintainable nor reusable. Although such solutions are recurring, their non-systematic implementation practices incur a significant software maintenance burden. Therefore, there is 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.

In this paper, we present 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++ [29], 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 woven with the original Fortran code.

Our 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 our approach is general and can benefit a substantial portion of scientific applications, we have 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.

We report on our 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. Our version, however, is more concise. On average, by using aspect inheritance we were able to reduce the amount of uncommented lines of hand-written code by as much as 27%. Thus, with our approach, the required adaptability functionality can be implemented more concisely, making it easier to maintain and reuse.

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

- *An approach to rejuvenate 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 reuse 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 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 2 summarizes common algorithmic level adaptation patterns in scientific computing. Section 3 outlines the complexities of engineering modern scientific software. Section 4 describes our systematic method for applying aspect-oriented abstractions to Fortran code. Section 5 describes how we implemented adaptation patterns as reusable aspects and applied them to a realistic scientific application. Section 6 evaluates the software engineering benefits of our approach and discusses limitations. Section 7 compares our approach to the related state of the art, and Section 8 presents concluding remarks.

2. ADAPTATION PATTERNS IN SCIENTIFIC COMPUTING

In this section, we present *adaptivity schemas*, common adaptation patterns of scientific computing, whose aspect-oriented implementation we describe in Section 5.

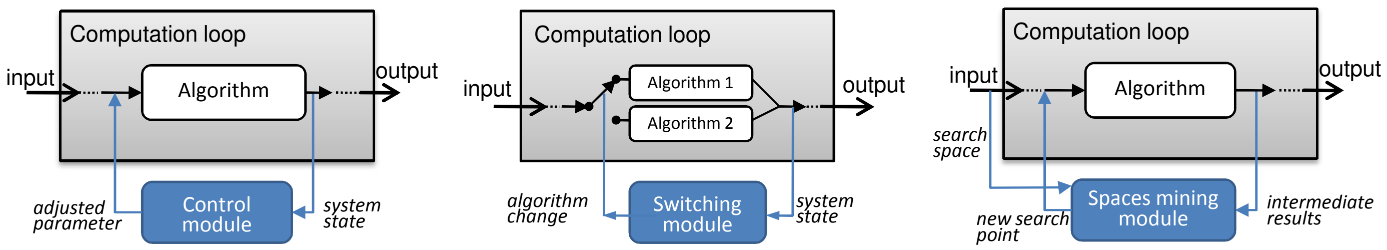
Since most execution time is spent on loops, they are a prime target of compiler optimization or parallelization techniques [6]. Adapting scientific programs also focuses on loops in most iterative computations—typically the end of a loop exhibits stable system state and consistent intermediate results. 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.

2.1 Overview of Adaptivity Schemas

Adaptivity schemas [31] 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.

2.1.1 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



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

Figure 1: Adaptivity Schemas

adaptation that adjusts the parameters to better match the dynamic characteristics of the computational progress. For example, Hovland and Heath [16] demonstrate how the relaxation parameter of the Successive Over-Relaxation (SOR) algorithm can be controlled through automatic differentiation. Figure 1a shows the schematic view of the Control Systems adaptivity schema.

2.1.2 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 [26] solver, used in ordinary differential equation 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.

2.1.3 Active Mining of Recommendation Spaces

An Active Mining of Recommendation Spaces schema describes 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 function is minimized.

3. ENGINEERING ADAPTABLE SCIENTIFIC SOFTWARE

Next, we shed some light on the realities of engineering modern scientific software.

3.1 Separating Concerns

The requirements imposed on modern scientific software are often so complex that they can be met only by means of a true collaboration between scientific programmers and computing experts. There are also complex social issues at play. Scientific programmers are often domain experts—scientists and engineers—who are extremely knowledgeable in their respective domains but may lack a deep understanding of computing or experience with modern developments in Software Engineering. In fact, scientific programmers are unlikely to be particularly enthusiastic about learning languages other than Fortran and to have familiarity with advanced software construction methodologies such as AOP. At the same time, the software engineers collaborating with scientific programmers to provide advanced adaptation functionality are likely to be eager to employ advanced software construction tools and techniques.

Thus, the approach described here aims at facilitating a productive collaboration of programmers from different communities. AOP becomes a technological solution to a set of social issues intrinsic to the construction of modern scientific applications. Specifically, our approach enables a smooth separation of concerns, allowing the core algorithmic functionality and its adaptivity schemas to be expressed in different languages by their respective domain experts. In other words, following our approach makes it possible for scientific programmers and computing experts to collaborate harmoniously while playing on their respective home turfs.

Using AOP in our context is a means of achieving a high degree of separating concerns. The fundamental concerns of modern scientific software are appropriately abstracted and separately implemented, and then assembled together in a highly adaptable scientific application.

3.2 Need for Portable Adaptations

The approach presented here was motivated by a real life scenario. Previously, we used a special-purpose adaptation library to implement the adaptivity schemas described above [18, 19]. Because the library’s implementation was anchored to 32-bit Unix environments, the adaptivity functionality was no longer available on 64-bit systems, which nowadays are standard platforms for the majority of scientific applications. It is this lack of portability that made us realize that more standardized and mainstream software engineering solutions must be introduced to obtain portable adaptations. Indeed, AspectC++ naturally supports both 32-bit and 64-bit architectures, so that the same adaptation source code can be applied to different architectures through a simple recompilation.

3.3 Complexity

Despite the known advantages of AOP in reducing the complexity of implementing cross-cutting concerns, specifying and reasoning about pointcuts can incur a significant burden on novice AOP programmers. Fortunately, the adaptation schemas we want to support tend to have quite straightforward pointcuts (e.g., initialization functions, parallel communication functions, etc.). Indeed, these join points tend to be quite intuitive. As such, they should be easily expressible not only by computing experts, but also by scientific programmers comfortable with using regular expressions.

4. ADAPTING FORTRAN PROGRAMS VIA C++ ASPECTS

A typical scientific application is written in Fortran and uses the Message Passing Interface (MPI) [11]. 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.

4.1 Integrating Fortran with AspectC++

Our approach leverages the binary compatibility between imperative languages compiled to the executable linkage format (ELF) [2]. 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++ equivalents, 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) 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.

4.2 Function Call Redirection

Once a desired portion of Fortran code is exposed via C wrappers to work as AspectC++ pointcuts, the function calls originally made to those wrapped functions in a given application need to be replaced with the calls to the corresponding C wrappers. Without this replacement or redirection of function calls, the C wrapper functions would not be called in the application, and thus the associated adaptation advice code would not be executed either.

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 [15, 24]. 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 make a call to the original implementation of `foo` instead of the wrapper.

An alternative function call redirection implementation can use LD_PRELOAD, an environment variable that specifies dynamic libraries (commonly referred to as dynamic shared objects (DSO)) 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). However, that implementation works only with shared objects, thus limiting its applicability.

4.3 Generating Fortran to C Wrappers

To implement a Fortran to C wrapper code generator, we extended F2PY [1], 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

```
! 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 2: The ITSOR subroutine and its C wrapper code generated by our wrapper generator

when exporting symbols to linkers; our wrapper generator creates function names accordingly.

Figure 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 3 illustrates the overall structure of our approach that weaves AspectC++ aspects with extant Fortran scientific applications. First, the needed pointcuts in a Fortran program are exposed as C functions through link-time wrapping and automated wrapper generation. The adaptation code expressed as AspectC++ advice structures is then woven at the exposed pointcuts. Finally, all the woven code is compiled and linked together with the original Fortran code, thus enhancing the original Fortran program with adaptive behavior.

5. 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 [30], 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 current version of GenIDLEST comprises about 78K lines of Fortran code located in 382 separate source files. The applications demonstrate how a Fortran scientific program can be adapted by woven adaptation code to enhance its capabilities in various aspects of simulation such as stability, accuracy, and performance.

5.1 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 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 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.

5.2 Control Systems Schema

Figure 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 [18] where we used a composition tool called Adaptive Code Collage (formerly called Invoke) to plug separately written adaptation code into GenIDLEST

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

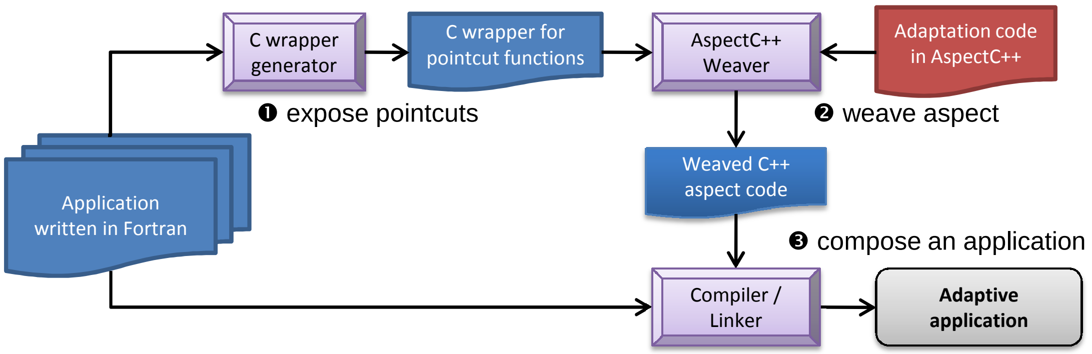


Figure 3: Weaving adaptation aspect code through AspectC++ by exposing Fortran functions in C wrappers

```
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: AspectC++ code for obtaining MPI execution environment information

```
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 5: Control systems schema implementation in AspectC++

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 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 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

```
#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 6: Timestep adaptation aspect to improve the stability of GenIDLEST simulations

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`).

5.3 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 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

```
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 7: Algorithm switching schema implementation in AspectC++

as we described in Section 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 need to provide a concrete definition for each of these functions.

The **`after`** advice executes switching operations after the `globalComm` pointcut function completes. 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

```
// 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 8: Flow model switching aspect to improve the accuracy of GenIDLEST simulations

implements flow model switching based on the `AlgoSwitching` aspect code.

Figure 8 shows the flow model switching implementation in AspectC++. The adaptation logic is similar to our previous work in [18], where a switching decision is dynamically 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`**.

5.4 Active Mining of Recommendation Spaces Schema

Figure 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

```
aspect Mining {
protected:
  bool srchComplete, explrComplete;
  DTYPE *srchPnt, *curPnt;
  unsigned int count, dim;
public:
  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

  // 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 9: Mining of spaces schema implementation in AspectC++

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 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 [19] 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 GenIDLEST preconditioning code, called cache sub-blocks,

```
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 10: Dynamic parameter tuning aspect to improve the performance of GenIDLEST simulations

is dynamically tuned through a staged optimization procedure to match the memory hierarchy of a given execution platform

Figure 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, the `evalExplore` function decides which one out of the two to use for the running computation and updates the parameter search space accordingly.

6. 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 [18, 19]. Finally, we discuss

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

some of the limitations of our approach.

6.1 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 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 adaptation 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 (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 2: Complexity comparison of the GenIDLEST adaptation implementations between the hand-coded and AspectC++ implementations using maximum MCC numbers

6.2 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) [23], a metric that directly measures the number of linearly independent paths through the program's source code. The MCC number is indicative of the effort the programmer has to expend to understand a codebase.

Table 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.

6.3 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

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

Table 3: Execution time (seconds) and overhead measurements of the GenIDLEST adaptation implementations using a pin fin array problem for 500 time steps.

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 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.

6.4 Limitations

The key limitations of our approach stem from the semantic differences between Fortran and C++, and to effectively adapt Fortran programs, we necessarily had to limit the subset of the Fortran language with which we want to be able to interface through AspectC++.

A current engineering limitation of our tool infrastructure is a lack of support for composite types. A Fortran composite data type is a global structure that can be mapped to some C++ global variable. As long as both the C++ and Fortran parts of an application conform to the ELF specification, a Fortran composite data type can be mapped to some C++ structure to ensure that both structures have the same layout. Thus, to use composite types with our approach, the programmer has to define an appropriate C++ counterpart for a Fortran complex data type, which can be tedious and error-prone if done manually. As a specific example, the Fortran `complex` data type does not have a native counterpart despite the presence of complex types in the C++ Standard Template Library.

We also had to carefully choose which features of AspectC++ we want to support. Our approach requires that only Fortran functions be exposed through wrappers to interface with C++ aspect code. We then leverage the AspectC++ **`execution`** pointcut through which the programmer can specify *callee-site* join points at the execution of wrapped Fortran function calls. The resulting callee-site weaving is easier to implement than *caller-site* weaving (e.g., the `call` pointcut); only one exposure point is required for each intercepted Fortran function, whereas the caller-site weaving has to examine the entire Fortran codebase to find every call-site for each intercepted function that must be exposed at all the call-sites. As a result, our callee-site weaving approach may be insufficiently powerful in the case of complex adaptations requiring complete context information, which may be unavailable from the exposed signature-only Fortran function data.

While AspectC++ supports other kinds of pointcuts, they would not be applicable for the needed adaptations. In addition, some of the AspectC++ pointcuts simply cannot be used due to the fine-grained differences in semantics between Fortran and C++, which restrict the range of applicability of certain AspectC++ features. For instance, AspectC++ offers class and namespace matching mechanisms to specify join points with the granularity of C++ classes or structures. However, it is not immediately obvious how one can apply them to Fortran, which does not have a direct counterpart to C++ classes.

As it turns out, even using a limited subset of AspectC++ features makes it possible to flexibly adapt Fortran programs at the function level, whereby separately developed Fortran and C++ programs work in concert to achieve a common goal of implementing adaptable scientific software.

7. RELATED WORK

In this section, we briefly survey related work in the literature of AOP and scientific computing, and contrast our work with them.

7.1 Multilingual Systems

Other approaches to integrating Fortran with C/C++ focused on language translation. For instance, there are Fortran 77 to C translators such as `f2c` [10]. Language interoperability tools include Chasm [27] and middleware such as Babel [22] and component technologies such as Common Component Architecture [4]. 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.

7.2 AOP for Scientific Computing

Although scientific computing was one of the initial application domains of AOP [17], 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 [3], 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. [13] tackle 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. [12] showcase 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. Aslam et al. [5] implement an aspect-oriented language for Matlab, a dynamic programming language popular in scientific programming. They apply their language to Matlab programs to implement typical AOP use cases such as performance profiling and data annotations. These AOP applications are contrasted with our work which aims to adapt program behavior to enhance an application's capabilities by tackling language interoperability issues.

7.3 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. Bangalore [7] uses AspectC++ to implement parallelization patterns and components such as data distribution and message passing on top of existing sequential programs, thus achieving modular development of parallel programs. Sobral [28] uses AspectJ for incrementally developing parallel applications with serial Java programs. Harbulot and Gurd [14] develop 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 [25] that uses compiler directives to express parallelism. For example, both the JBOSS AOP framework[2] and recent versions of AspectJ [3] provide the `@Oneway` annotation to fire **`void`** methods in a separate thread that will run asynchronously in a task-parallel fashion. The aspect-based approach in [8] is similar to ours 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.

[2]http://www.jboss.org/jbossaop
[3]http://www.eclipse.org/aspectj

8. CONCLUSION

In this paper, we presented a novel approach that expresses 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. Future work directions will focus on providing a more complete library of adaptivity schema aspects to support additional adaptivity schemas [31].

Facing the unprecedented challenges of modern scientific applications requires the adoption of state-of-the-art software engineering techniques and approaches. In that light, the maintainability advantages offered by AOP can offer viable solutions to these challenges. The ideas presented in this paper can inform the designs that transfer the lessons gleaned from constructing and maintaining mainstream software systems to help address the challenges of emerging scientific software.

9. ACKNOWLEDGMENTS

The authors would like to thank Dr. Danesh K. Tafti in the department of mechanical engineering at Virginia Tech for his help in letting us use his GenIDLEST CFD simulation software for this research. We acknowledge the support of US National Science Foundation grant CNS-0615181.

10. REFERENCES

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

[2] Tools Interface Standards (TIS) Committee. Executable and Linking Format (ELF) Specification, 1995.

[3] B. Amedro, V. Bodnartchouk, D. Caromel, C. Delbe, F. Huet, and G. Taboada. Current State of Java for HPC. Technical Report RT-0353, INRIA, 2008.

[4] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. 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.

[5] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren. AspectMatlab: An Aspect-Oriented Scientific Programming Language. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented*

Software Development, pages 181–192, Rennes and Saint-Malo, France, 2010.

[6] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[7] P. V. Bangalore. Generating Parallel Applications for Distributed Memory Systems using Aspects, Components, and Patterns. In *ACP4IS '07: Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Vancouver, British Columbia, Canada, 2007.

[8] C. A. Cunha, J. L. Sobral, and M. 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, Bonn, Germany, 2006.

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

[10] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. A Fortran to C converter. Technical Report 149, AT&T Bell Laboratories, 1995.

[11] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] H. Han, H. Jung, H. Y. Yeom, and D.-Y. 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.

[13] B. Harbulot and J. 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, Lancaster, UK, 2004.

[14] B. Harbulot and J. 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, Bonn, Germany, 2006.

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

[16] P. D. Hovland and M. 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.

[17] J. Irwin, J.-M. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. 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.

[18] P. Kang, N. K. C. Selvarasu, N. Ramakrishnan, C. J. Ribbens, D. K. Tafti, and S. Varadarajan. Modular, Fine-Grained Adaptation of Parallel Programs. In *Proceedings of the 9th International Conference on Computational Science*, pages 269–279, Baton Rouge, Louisiana, USA, May 2009.

[19] P. Kang, N. K. C. Selvarasu, N. Ramakrishnan, C. J. Ribbens, D. K. Tafti, and S. Varadarajan. Dynamic Tuning of Algorithmic Parameters of Parallel Scientific Codes. In *Proceedings of the 10th International Conference on Computational Science*, pages 145–153, Amsterdam, The Netherlands, May 2010.

[20] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer-Verlag, Finland, June 1997.

[21] D. K. Kim, Y. Jiao, and E. 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.

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

[23] T. 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.

[24] D. S. Myers and A. 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.

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

[26] 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.

[27] C. E. Rasmussen, M. J. Sottile, S. S. Shende, and A. D. Malony. Bridging the Language Gap in Scientific Computing: the Chasm Approach. *Concurrency and Computation: Practice & Experience*, 18(2):151–162, 2006.

[28] J. Sobral. Incrementally Developing Parallel Applications with AspectJ. *IPDPS '06*, 0:95, 2006.

[29] O. Spinczyk, A. Gal, and W. 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.

[30] D. Tafti. GenIDLEST - A Scalable Parallel Computational Tool for Simulating Complex Turbulent Flows. In *Proceedings of the ASME Fluids Engineering Division*, volume 256. ASME-IMECE, Nov. 2001.

[31] S. Varadarajan and N. Ramakrishnan. Novel Runtime Systems Support for Adaptive Compositional Modeling in PSEs. *Future Gener. Comput. Syst.*, 21(6):878–895, 2005.