The Adaptive Code Kitchen: Flexible Tools for Dynamic Application Composition

Pilsung Kang[1], Mike Heffner[1], Joy Mukherjee[1],
Naren Ramakrishnan[1], Srinidhi Varadarajan[1], Cal Ribbens[1], and Danesh K. Tafti[2]
[1]Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, USA
[2]Department of Mechanical Engineering, Virginia Tech, Blacksburg, VA 24061, USA

Abstract

Driven by the increasing componentization of scientific codes, the deployment of high-end system infrastructures such as the Grid, and the desire to support high level problem solving primitives, application composition systems have become prevalent in computational science practice. We present the adaptive code kitchen which, as the name connotes, is a loose collection of capabilities to help realize complex adaptive composition scenarios. These include function interception, continuation modification, dynamic process checkpointing and rollback, and runtime recommendation. Using these broad primitives, a computational scientist can specify many 'recipes' of adaptivity as complete control systems around native object codes. Runtime systems support then enables loading and linking of native code components, monitoring of performance indicators, consulting a recommender system for algorithmic decisions, and dynamically updating application components in response to the recommendations. We present the architecture of the adaptive code kitchen and the key enabling technologies with brief mention of the applications that will be investigated henceforth during the course of the project.

1. Introduction

Fueled by rapid advances in high-end computing infrastructures [14] and problem solving environments (PSEs) [28], application composition systems [26] are now prevalent in computational science practice. Diverse areas, from nuclear physics, to earth science simulation, to bioinformatics now benefit from compositional specification and realization of large-scale computations. Such systems not only enable the view of scientific applications as a collection of distributed executable components but can also reactively respond to runtime considerations, e.g., by online substitution of code modules or dynamic re-wiring of application logic. The driving motivation is to make composition systems cognizant of constantly changing application needs and resource constraints [1, 19].

The scope of adaptive composition today ranges from manual, user-guided querying and steering of computational runs [4] to automatic recommender systems that mine performance data [27], respond to runtime events, and substitute algorithms and models suitably. In well-structured domains of scientific software, such as linear algebra [12], accurate performance models of solvers over a range of problem characteristics, memory hierarchies, and architectures are available which can be harnessed to guide composition decisions. Once a decision to adapt is made, how to actually perform the adaptation has been investigated from the perspectives of compiler support for scientific computing, adaptive programming, and tunability interfaces. Some approaches, e.g., [7, 8], work at the pre-compiler level and provide adaptivity around programmed breakpoints by use of locks and barriers in parallel codes. Others, e.g., [13], specify adaptivity decisions in terms of operations on intermediate representations [2], also relying on compiler technology to help abstract distributed program behavior. The adaptive programming literature views adaptivity as a crosscutting concern, akin to aspect-oriented programming [20], and aims to provide flexible join points between modules, supporting remapping and manipulation of function invocations. Yet another approach is the use of an external tunability interface, as done in [11], that employs annotations to indicate alternate execution paths and adaptation controls.

A key determinant of the success of composition systems is the amount of buy-in solicited, e.g., commitment to a particular programming paradigm or modeling methodology, limitations on the forms of adaptivity that can be supported, or restrictions on the domain of applicability. We envision adaptive composition [34] as supporting componentization over native object codes (language neutrality),

1-4244-0910-1/07/$20.00 ©2007 IEEE.

providing arbitrary code expansion, contraction, and substitution operations (flexible adaptation), and incorporating runtime recommender systems for algorithm selection (model-based control). Existing solutions typically emphasize one or at most two of these aspects and do not provide an integrated environment to explore adaptivity scenarios over arbitrary scientific codes. In particular, there is often a tradeoff between the sophistication of decision support versus the sophistication of systems support. Approaches that provide expressive runtime support have traditionally focused on simple forms of adaptivity decisions, involving resource configurations and partitioning strategies. Whereas approaches that can explicitly model computations at the level of algorithms and models can indeed perform intelligent dynamic decisions but typically assume a particular runtime operating environment.

This paper introduces the *adaptive code kitchen* which, as the name connotes, is a loose collection of capabilities to help realize complex adaptive composition scenarios. These include function interception, continuation modification, dynamic process checkpointing and rollback, and runtime recommendation. Using these broad primitives, a computational scientist will be able to specify many 'recipes' of adaptivity as complete control systems around native object codes. We posit a broad picture of adaptivity here, one which is not restricted to identifying partitioning parameters, modifying data decompositions, or parallel scheduling; instead adaptivity is proposed at a more logical unit of algorithms and object codes. A cookbook of standard adaptivity schemas will be developed, besides facilities for users to explore their individual approaches to dynamically reconfigure applications. Runtime systems support will then enable loading and linking of native code components, monitoring of performance indicators, consulting a recommender system for algorithmic decisions, and dynamically updating application components in response to the recommendations.

2. Application Context

We study code composition research issues in the context of the multi-physics, multi-scale domain of turbulence simulation, primarily for the many opportunities for adaptive composition that it affords, and for its relevance to progress in areas such as atmospheric simulation, aerodynamics, and earth system science. Multi-physics, multi-scale applications are by nature dynamic, owing to changing problem characteristics, the need to spawn auxiliary calculations, automatic error estimation, and changing loads in a parallel distributed environment, among other factors. The simulation of turbulence, especially, is one such application encapsulating length and time scales over several orders of magnitude. Turbulence manifests in the atmosphere, oceans, and man made devices which collectively have a large economic impact. Hence the accurate prediction of such flows can have a significant impact on societal well-being. For instance, a 10% turbulent drag (friction) reduction in airplanes, ships, and automobiles can have a major impact on energy consumption.

The current state-of-the-art consists of several methods and (multi-scale) combinations of them with varying degrees of approximations and modeling. For instance, a zeroth-order modeling is performed by solving the Reynolds-averaged Navier-Stokes (RANS) equations in which only the mean effect of turbulence is taken into consideration. On the other end of the computational spectrum are direct numerical simulations (DNS) in which all the scales are resolved. While RANS modeling is a widely used engineering tool because of its affordability, it suffers from large uncertainties in complex non-canonical flows. DNS, being physically accurate, has been used successfully in simple geometries to advance the fundamental understanding of turbulence. However, it is prohibitively expensive and quickly becomes intractable as the flow Reynolds number increases and, in complex engineering geometries with current day computing power, is projected to remain so for decades to come. Hence other methods whose complexities lie between the two extremes have been devised. Large eddy simulations (LES) resolve the important scales of turbulence and model the smaller dissipative scales which are more universal and easier to model. In this domain, it is not uncommon for each calculation to take several weeks to a month or two for one computation to complete on several hundred processors. Thus even a small gain in improving the efficiency of the computation can be quite beneficial.

Typically a full simulation is composed of different components starting from the choice of grid, algorithms, numerical discretizations, turbulence models, stability criteria, solution of linear systems, all of which have an influence both on the accuracy and time-to-solution. While it is obviously advantageous to employ the most efficient composition, the interaction between components is typically not known *a priori* and often changes during the course of the computation. For instance, in an evolving turbulent flow, the properties of the linear system may change over the course of a simulation and using the same solver settings for the whole simulation may be sub-optimal.

GenIDLEST [32] is a scalable parallel computational framework developed by co-PI Tafti for the simulation of turbulence in complex geometries. It can be used in various capabilities ranging from DNS to RANS simulations with auxiliary models for RANS, LES, URANS (unsteady RANS) and detached-eddy simulations (DES) which uses a hybrid multi-scale RANS-LES approach. The development effort has spanned over a decade and the code is robust and validated in a number of flows ranging from high

Reynolds number turbulence to electrokinetic microflows. GenIDLEST uses a combination of F77 codes at its core with F90 wrappers for dynamic memory management; it is instrumented for use with both MPI and OpenMP parallelism for distributed as well as shared memory architectures and has a range of features for numerical discretizations, algorithms, boundary conditions, linear solvers (both indirect and direct), and post-processing of data. We ground our work by enabling the adaptive code kitchen capabilities to compose solvers within the GenIDLEST framework.

3. Research Goals

To realize our vision of a flexible code composition framework, the adaptive code kitchen focuses on a set of five inter-related aspects.

1. **An integrated framework for composing object codes, including legacy software.** The vast majority of scientific codes are written in procedural languages such as FORTRAN and C, compiled into collections of individual object files with specified binding interfaces. These object files must be directly composable in our framework and must be organizable into groups of concerted programs alongside task decompositions and data exchanges. Furthermore, the framework must support the dynamic insertion of new codes into a running application. For instance, during the course of a long running turbulence simulation, a new solver might become available, and must be substitutable for the currently active solution method, without interrupting the computation.

2. **A runtime system capable of instrumenting function interception, continuation modification, and dynamic process checkpointing and rollback into a composed (and running) parallel application.** Given a composed application, we must provide rich primitives to harness the state of the running program as defined by key application parameters, domain variables, and other qualitative indicators of interest (e.g., 'has the Reynolds number changed?'). Checkpointing capabilities must be specifiable around function invocation boundaries and dynamic rollbacks predicated on problem properties will be required (e.g., 'rollback to the earliest point when the matrix was not so ill-conditioned').

3. **A runtime recommender system for modeling adaptive control as sequential decision making problems with ability to generalize from observed performance.** A recommender system models a sequential decision making problem, i.e., of all the decisions made in the current computational run, which are contributing to the success or failure of the run? A recommender maintains the utility of algorithmic choices (actions) as a function of problem parameters and other indicators (states). To perform credit (or blame) assignment, a recommender must balance the twin considerations of *exploration* and *exploitation*. Exploration is required in order to see whether there exists a better algorithmic choice than what is currently considered to be the best. Exploitation is when we must harness the performance data gathered thus far to make optimal algorithm choices. As argued eloquently in [31], we cannot pursue either exploration or exploitation exclusively without failing at it, and hence a recommender system must judiciously balance the two objectives. In addition, a recommender system must generalize from problems solved so far to problems that have not been encountered before.

4. **A library of adaptivity schemas specifying skeletons of how composition and adaptation will occur.** The primitives must be organized into schemas of adaptivity, giving rise to a diversity of composition possibilities. We can think of adaptivity in a temporal as well as spatial sense; temporal when the flow of control is predetermined but the application can retrace its path and redo steps with different choices, spatial when new flows of control can be dynamically introduced. Together, it must be possible to realize a complete control system (e.g., a PID controller) over the state variables defined by an object code. Given an adaptivity schema specification, we must be able to breakdown the specification into aspects of what to checkpoint, where to checkpoint, when to instrument rollbacks, and how to measure progress.

5. **Interfaces for human-in-the-loop control of executing scientific codes.**

Finally, to impact large-scale scientific applications, we will provide support for codes executing on both shared-memory as well distributed-memory platforms. Needless to say, checkpointing and consistency issues are amplified here. We require the use of a database for both recording snapshots of a running computation as well as summarizing performance data from past computations in generalized form. Operational details of achieving composition, e.g., reasoning about impedance mismatches in function invocation formats, are also important.

4. Enabling Technologies

Two key enabling technologies, also supported by the NSF Next Generation Software Program, provide the building blocks necessary for the adaptive code kitchen:

h

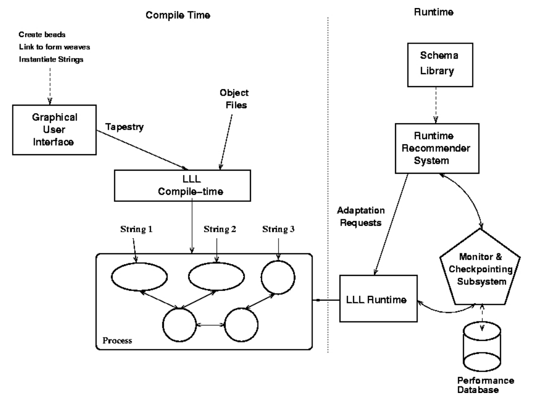

Figure 1. The system architecture of the adaptive code kitchen includes both compile-time and runtime elements. A composer uses the GUI to compose modules into an application which can then be adapted at runtime by a runtime recommender.

Weaves, a source-language independent parallel framework for object-based composition of unmodified scientific codes, and recommender systems that mine performance databases from algorithmic executions to make dynamic decisions about algorithms and models.

4.1 Object Based Composition

The Weaves framework [22], funded in part by NSF CAREER grant EIA-0133840, creates a new language independent abstraction for object-based composition of unmodified code modules. Weaves works through reverse-compiler analysis of compiled object files, enabling the vast repository of legacy scientific libraries to be seamlessly used in a object-based compositional framework, *without requiring that these codes be written in an object-oriented language*.

For the purposes of this project, Weaves can create multiple instantiations (called **beads**) of code modules similar in principle to OOP's objects and classes, which can then be composed. Formally, a **module** is any object file or collection of object files defined by the user. Modules have a data context (the global state of the module scoped within the object files of the module) and a code context (with potentially multiple entry point and exit point functions). A bead is an instantiation of a module; multiple instantiations of a module have independent data contexts, but share the same code contexts. A **weave** is a collection of data contexts belonging to beads of different modules. Unlike processes which have a single namespace mapped to a single address space, Weaves thus allows users to define multiple namespaces within the same address space. A **string** is a thread of execution that operates within a single weave. Similar to the threads model, multiple strings may execute within a single weave. However, a single string cannot operate under multiple weaves at the same time. Finally, a **tapestry** is a set of weaves, which describes the structure of the composed application. The physical manifestation of a tapestry is typically a single process. The design of Weaves bears a strong resemblance to the OO framework of Mentat proposed in [15]; however, Mentat requires creating code objects in an OO language wheres Weaves can create an object based framework from code written in any language.

Fig. 1 depicts the design process in the Weaves framework and how it dovetails with the rest of the elements of the adaptive code kitchen. The design process involves two entities: a *programmer* who implements the modules (shown as object files in Fig. 1) and a *composer*, who uses a graphical user interface to instantiate beads and define the various weaves and strings. The result of the GUI composition is a tapestry configuration file, which is used to load and execute the composed application (represented as 'LLL compile-time' in Fig. 1). Each composed application also has a module called a monitor that is automatically linked with the composed application, and provides a powerful run-time interface to query the state/statistics of the composed application and modify its structure by creating new beads, defining weaves, and instantiating strings at runtime. The Weaves framework has reached prototype capability, supporting the vast majority of UNIX platforms and language compilers that use the ELF format. While Weaves originated as a compiler technology to support scalable network emulation (the focus of grant EIA-0133840), it has now been extended to support compositional modeling of scientific codes. These applications include the Sweep3D benchmark for discrete-ordinates neutron transport, collaborating ELLPACK partial differential equation solvers, and checkpoint and run-time migration of parallel grid applications. For more details refer to [22, 34].

4.2 Runtime Recommender Systems

Recommender systems aid in the automatic or semi-automatic selection of algorithms in a PSE; for instance, given a problem in numerical quadrature and performance constraints on its solution, a recommender system helps select the best (or nearly best) algorithm [25]. Recommender systems are very useful when domain knowledge is imperfect and where our understanding of the factors influencing algorithm applicability is incomplete. One of the main research issues here is understanding the fundamental mechanisms by which knowledge about scientific problems and algorithms is created, validated, and communicated. A typical approach to designing recommender systems is to first organize a database of problem populations and algorithm executions, and subsequently mine this database to understand the selective superiority of algorithms. Supported in part by NSF CAREER grant EIA-9984317, this idea has been recently extended in many important ways – mining recommendation spaces with continuous-valued attributes, a recommendation portal with database and experiment management support for performance data analysis, and automatic mining of recommendation spaces, providing support for algorithm selection at runtime, knowledge-based compositional modeling, and harnessing domain knowledge of physical properties underlying problems. See [34] for more details.

There are two main approaches to prototyping recommender systems. In the *offline* approach, we organize a benchmark database of realistic test problems and algorithm executions, mine this database to generalize from the performance data, and use the results of mining to guide selection of appropriate algorithms and software (for future problems). The end-goal is to use supervised machine learning and data mining algorithms to empirically capture the relationship between problem characteristics, algorithm parameters, and algorithm performance [33]. Assuring adequate coverage of the training dataset and carefully generalizing to new problems are crucial here. In the *online* approach, the data collection phase is interleaved with the mining process, so that we can proactively choose problems that will accurately and efficiently sample desired regions of the recommendation space. In this approach, algorithms based on reinforcement learning [31] are especially useful. As mentioned earlier, the *exploration* phase of these algorithms judiciously samples new recommendation choices to improve their learning, and the *exploitation* phase is used in a production context (i.e., to drive real-time selection of algorithms). In addition to the issues of dataset coverage and generalization, these algorithms must also perform sequential credit assignment in a cascading series of decisions.

5. Building the Adaptive Code Kitchen

The three main research tasks studied currently are to be able to provide componentization over native object codes, support arbitrary code expansion, contraction, and substitution operations, and incorporate runtime recommender systems in model-based control scenarios. Each of these tasks is elaborated upon below.

5.1 Load and Let Link

The first research task underlying the adaptive code kitchen to develop a framework for agile and flexible runtime loading and composition of native code components. Just as object files are linked at compile-time to generate a program executable, we seek to link modules at runtime to obtain a program image (denoted as 'LLL runtime' in Fig. 1). In doing so, we seek to maintain the reified structure inherent in an executable's constituent object files even after runtime composition. Furthermore, to support arbitrary code-base expansion and contraction [29], we must be lenient toward undefined references, and provide a default flow of control that may be used to monitor and asynchronously modify the composition.

Our solution approach is to extend the Weaves framework to allow runtime control over addition and modification of individual functions and variables in modules, which is instrumental in letting applications dynamically add, prune, or modify their functionality, content, and structure. Rather than rigorously require resolution of all references for successful completion, we accommodate undefined or, more aptly, dangling references, which is conducive to arbitrary code-base expansion and contraction. Applications may thus specify their own runtime handlers to switch references and tune dynamic composition to their needs. This is similar in spirit to late-binding mechanisms [23] except that handlers may be explicitly specified by the application, may be different for different references, and (safely) default to undefined values. In addition, our minimal definition of modules allows the framework to flexibly work with other high-level frameworks and models for reification and reflection [9, 24, 30]. Some aspects of our mechanism may be even more productive when coupled with strategies for dynamic state and stack manipulation [16, 17].

It is important to contrast what is proposed here against traditional binary loaders (GNU `ld`) which are typically monolithic systems-level libraries. The interfaces they expose and the functionalities they offer are not enough to support the flexibility demands of the adaptive code kitchen. Contemporary late binding mechanisms for binaries tolerate dangling functional references only as long as they are not accessed. But when such accesses do happen in absence of compatible definitions, they result in anomalous behavior (further, dangling references to data elements are not allowed). In order to maintain the modular structure at runtime, we require that objects be loaded separately before application composition. This can result in some modules being loaded with several dangling references not just to functions but also to data elements. Again, typical loaders do not provide an explicit interface to connect a reference to an arbitrary definition. Hence, we will design our own tool—'load and let link (LLL)'—for dynamic loading and linking of modules that offers extra control through simple interfaces (APIs).

We have designed a preliminary prototype implementation of LLL [21] that works on 32-bit x86 architectures running GNU/Linux. As discussed previously, it relies heavily on the executable and linkable file format (ELF) used by most UNIX systems. It recognizes shared object (`.so`) files as loadable modules. Shared objects can be easily created from most relocatable objects (`.o`) compiled with position independent options (e.g., `-fPIC` for `gcc`). Using shared objects with position independent code can result in some extra instructions vis-a-vis normal compilation. However, the associated performance and memory overheads are as low as in regular dynamic libraries. LLL currently can be customized to use either POSIX threads (pthreads) or GNUs user-level threads package (Pth), and a port to the x86_64 architecture is currently underway.

5.2 Primitives for Runtime Composition

The second research task envisaged here is to provide broad primitives for realizing adaptive composition scenarios. Before we present the functional aspects, it is important to point out that the ELF instrumentation techniques that are at the core of the Weaves framework can, to a large extent, automate the process of analyzing native code objects (modules). Such analysis will be crucial to the engineering of adaptivity schemas later (by unraveling certain source level details without requiring the source codes).

5.2.1 Function Interception

One of the basic goals of our framework is to support procedure-level decomposition of a compiled object file so that procedure calls within a module become control points at which the application's execution can be steered. Therefore, the basic construct required in the framework is a method for intercepting, or *catching*, the function calls made within an application. There are a variety of programs and projects that support intercepting function calls within an application [5, 18]. Our framework is different from these projects in that it intercepts function calls at the location the calls are made. Through our LLL facility, this primitive will wrap any desired function call with pre- and post-handlers. These handlers become programs that capture dynamic state snapshots of the application and can even determine whether to continue execution with the originally composed application or to perform some other computation.

It is imperative that function interception be implemented at the caller end, since the callee might be located in a dynamic module that is not currently available. The original function call location in effect becomes a placeholder (or "link point," in the vernacular of the aspect oriented programming literature) that the framework can connect to arbitrary procedures during runtime. Similarly, the interception mechanism can divert function returns as well using return handlers that can perform various housekeeping duties, before eventually returning to the original caller. This is useful when the outcome of the function call must be queried, including the return value or any values returned via pass-by-reference parameters. This allows us to massage any return values before they are passed back to the calling module to, for example, fix type differences or influence the caller (via a recommender).

5.2.2 Continuation modification

When a function call is intercepted or the return of a function is intercepted, the framework could pass control of the application to the recommender system so that it may make any required adaptive control decisions. We envisage the recommender as registering either pre- or post-callbacks for desired functions, to be triggered whenever these function symbols are invoked or a return is made from them. Given a reference to the current invocation stack entry in the framework, the recommender system can: lookup or manipulate parameters through ELF analysis, remap the function call, search the remaining invocation stack, and checkpoint or rollback the process. In other words, we have expressive access to the current state of the application and can modify continuations suitably. For instance, all references to a direct solver can be changed into references to an iterative solver based on a runtime consistency check.

We also envisage a complete set of controls that allow the application to query and manipulate the parameters passed to procedure calls. The instantaneous values of parameters passed to a function at runtime can give insight into whether an application is making gainful progress or to whether any adaptive decisions should be made to improve the results or performance. For example, tracking the current relative error of a numerical routine might reveal characteristics of the problem that suggest that switching to a different routine might improve convergence.

5.2.3 Dynamic process checkpointing and rollback

The most valuable primitive studied here is the ability to arbitrarily 'rollback' the execution of an application to a previous state. Scenarios requiring this capability are covered later. Our framework utilizes the DejaVu library, a distributed snapshot and fault tolerance library for performing dynamic process rollback. DejaVu, supported by ongoing NSF grant CNS-0325534, supports online, incremental, process checkpointing of all static and dynamic memory and all file and network I/O. A function symbol can be registered with our framework to be checkpointed so that every time an invocation of that function occurs, a checkpoint is saved just prior to executing the function. Function invocations are not checkpointed by default because to do so at such a fine granularity is expensive. Instead, the recommender should determine an appropriate mix between coarse-grained and fine-grained checkpointing relative to the application. Each checkpoint operation returns a unique identifier that is stored within the respective function invocation record. This identifier can then be used to support predicate-based rollback and recovery, e.g., to rollback to the earliest cataloged instance when a given variable was within bounds.

An important criteria for a process checkpointing library used in an adaptive computing framework is that it must support the ability to maintain 'non-volatile' memory regions. Therefore, within the DejaVu library there is support for handling partial memory rollbacks. A dedicated memory allocator within DejaVu creates memory regions whose contents are not reverted during a rollback. This feature is important in, for instance, maintaining the entire function invocation stack so that the recommender can implement remappings after rolling back to a starting state.

The above approach to checkpointing directly carries over to shared-memory and distributed shared memory paradigms. In cluster computing environments, a high degree of problem decomposition across processors requires some level of care in implementing checkpointing mechanisms. Our approach here is to insert a barrier in the function interception pre-handlers so that all processes synchronize around this point, at which time each process can instrument a local checkpoint. Truly asynchronous distributed checkpointing mechanisms that maintain global consistencies are an active area of research and outside the scope of this project.

5.3 Adaptivity Schemas

The capabilities described thus far endow a recommender system with potential to serve as a full-fledged controller for the algorithmic solution of scientific problems. As illustrated in Fig. 1, we can view the recommender as an overseer of the computation, either passively monitoring choices made (by existing compositions) and learning the utilities of algorithms for different states or actively making exploratory choices to learn about algorithm superiority. The former requires only the use of the function interception primitive, whereas the latter requires continuation modification and (potentially) dynamic process rollback.

An adaptivity schema is a skeleton that specifies the control points at which the recommender can observe and influence the composition but does not specify the actual choices themselves. A schema thus suggests a high-level mode of problem solving [3], not unlike parameter sweeps [10] and algorithmic bombardment [6], and is pertinent in specific application contexts. A selection of adaptivity schemas has been presented elsewhere [34] and hence we do not elaborate on these further in this paper.

6. Ongoing Work

The basic LLL and runtime composition primitives have been implemented and applications of these in the GenIDLEST context are underway. In the context of turbulence simulation, we envisage the use of the adaptive code kitchen at three basic levels. At the algorithmic level, there are a multitude of possibilities depending on the type of flow: steady flow, time-dependent flow, compressible high-speed flow, incompressible low-speed flow, and variable density flow. At the model level, there are again numerous possibilities that describe the behavior of turbulence which is not resolved by the calculation (RANS, LES, and DES). Finally, at the solver level, linear system solution accounts for about 80-90% of the total computational time in the computations planned here. In dynamic computations (moving grids or changing physics) the characteristics of the linear system may also change with time. We plan to materialize the adaptive schema skeletons from [34] to address all the above opportunities and use the realized compositions to simulate leading edge film cooling flows for gas turbine blades.

Acknowledgements

This research was funded by NSF CSR grant CNS-0615181.

References

[1] V. Adve, A. Akinsanmi, J. Browne, D. Buaklee, G. Deng, V. Vi Lam, T. Morgan, J. Rice, G. Rodin, P. Teller, G. Tracy, M. Vernon, and S. Wright. Model-based Control of Adaptive Applications: An Overview. In *Proceedings of the Next Generation Software Systems Program Workshop (NGS), 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 2002. 8 pages.

[2] V. Adve and R. Sakellariou. Application Representations for Multiparadigm Performance Modeling of Large-Scale Parallel Scientific Codes. *International Journal of High Performance Computing Applications*, Vol. 14(4):pages 304–316, Winter 2000.

[3] M. Aldinucci, S. Campa, M. Coppoloa, M. Danelutto, C. Zoccolo, F. Andre, and J. Buisson. Parallel Program/Component Adaptivity Management. In *Proceedings of the CoreGRID Integration Workshop*, 2005.

[4] G. Allen, W. Benger, T. Goodale, H. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. Cactus Tools for Grid Applications. *Cluster Computing*, Vol. 4(3):pages 179–188, 2001.

[5] R. Balzer and N. Goldman. Mediating Connectors: A Non-ByPassable Process Wrapping Technology. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Middleware Workshop*, pages 73–77, 1999.

[6] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, and C. Romine. Algorithmic Bombardment for the Iterative Solution of Linear Systems: A Poly-Iterative Approach. *Journal of Computational and Applied Mathematics*, Vol. 74(1–2):pages 91–109, 1996.

[7] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. C^3: A System for Automated Application-Level Checkpointing of MPI Programs. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, pages 357–373, 2003.

[8] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-Level Checkpointing for Shared Memory Programs. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPOLOS-XI)*, pages 235–247, 2004.

[9] L. Capra, W. Emmerich, and C. Mascolo. Middleware for Mobile Computing: Awareness vs. Transparency. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001. Schloss Elmau, Germany.

[10] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of Supercomputing 2000*, Nov 2000. Dallas, TX.

[11] F. Chang and V. Karamcheti. A Framework for Automatic Adaptation of Tunable Distributed Applications. *Cluster Computing*, Vol. 4(1):pages 49–62, 2001.

[12] V. Eijkhout, E. Fuentes, T. Eidson, and J. Dongarra. The Component Structure of a Self-Adapting Numerical Software System. *International Journal of Parallel Programming*, Vol. 33(2–3):pages 137–143, Jun 2005.

[13] B. Ensink, J. Stanley, and V. Adve. Program Control Language: A Programming Language for Adaptive Distributed Applications. *Journal of Parallel and Distributed Computing*, Vol. 63(11):pages 1082–04, Nov 2003.

[14] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, Nov 2003.

[15] A. Grimshaw, J. Weissman, and W. Strayer. Portable Runtime Support for Dynamic Object Oriented Parallel Processing. *ACM Transactions on Computer Systems*, Vol. 14(2):pages 139–170, May 1996.

[16] M. Heffner. A Runtime Framework for Adaptive Compositional Modeling. Master's thesis, Department of Computer Science, Virginia Tech, 2004.

[17] G. Huang, H. Mei, and Q.-X. Wang. Towards Software Architecture at Runtime. *ACM SIGSOFT Software Engineering Notes*, Vol. 28(2):page 8, Mar 2003.

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

[19] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, R. Aydt, D. Reed, D. Gannon, S. Johnsson, C. Kesselman, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a Framework for Preparing and Executing Adaptive Grid Programs. In *Proceedings of the Next Generation Software Systems Program Workshop (NGS), 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 2002. 5 pages.

[20] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, Vol. 44(10):pages 39–41, 2001.

[21] J. Mukherjee and S. Varadarajan. Develop Once, Deploy Anywhere: Achieving Adaptivity with a Runtime Linker/Loader Framework. In *Proceedings of the Fourth Workshop on Reflective and Adaptive Middleware Systems*, pages 1–6, 2005. Grenoble, France.

[22] J. Mukherjee and S. Varadarajan. Weaves: A Framework for Reconfigurable Programming. *International Journal of Parallel Programming*, Vol. 33(2–3):pages 279–305, June 2005.

[23] D. Orr, J. Lepreau, J. Bonn, and R. Mecklenburg. Fast and Flexible Shared Libraries. In *Proceedings of the Summer USENIX Conference*, 1993.

[24] D. Pierre-Charles and T. Ledoux. Towards a Framework for Self-Adaptive Component-Based Applications. In *Proceedings of the Fourth IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03)*, Nov 2003. Paris, France.

[25] N. Ramakrishnan, J. Rice, and E. Houstis. GAUSS: An Online Algorithm Selection System for Numerical Quadrature. *Advances in Engineering Software*, Vol. 33(1):pages 27–36, Jan 2002.

[26] N. Ramakrishnan, L. Watson, D. Kafura, C. Ribbens, and C. Shaffer. Programming Environments for Multidisciplinary Grid Communities. *Concurrency and Computation: Practice and Experience*, Vol. 14(13–15):pages 1241–1273, Nov-Dec 2002.

[27] R. Ribler, H. Simitci, and D. Reed. The AutoPilot Performance-Directed Adaptive Control System. *Future Generation Computer Systems*, Vol. 18(1):pages 175–187, Sep 2001.

[28] J. Rice and R. Boisvert. From Scientific Software Libraries to Problem Solving Environments. *IEEE Computational Science and Engineering*, Vol. 3(3):pages 44–53, Sep 1996.

[29] P. Rigole, Y. Berbers, T. Holvoet, and K. Leuven. Mobile Adaptive Tasks Guided by Resource Contracts. In *Proceedings of the Second Workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC'04)*, Oct 2004. Toronto, Canada.

[30] B. Smith. *Reflection and Semantics in a Procedural Programming Language*. PhD thesis, Massachusetts Institute of Technology, Boston, 1982.

[31] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, 1998.

[32] D. 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, Nov 2001.

[33] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. Amato, and L. Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 2005)*, pages 277–288, June 2005. Chicago, IL.

[34] S. Varadarajan and N. Ramakrishnan. Novel Runtime Systems Support for Adaptive Compositional Modeling in PSEs. *Future Generation Computer Systems*, Vol. 21(6):pages 878–895, 2005.