Adaptive Code Collage: A Framework to Transparently Modify Scientific Codes

Legacy scientific codes are often repurposed to fit adaptive needs, but making such code adaptive without changing the original source programs can be challenging. Adaptive Code Collage (ACC) meets this challenge using function-call interception in a language-neutral way at link time, transparently "catching" and redirecting function calls.

Adaptive programs change their runtime behavior to achieve certain purposes—such as performance improvement, or better numerical stability—by tracking changes in problem characteristics, available resources, and execution environments. Almost every computational scientist has encountered the situation wherein they'd like to make a legacy code adaptive to their application context but without seriously changing the original programs. Simple examples of adaptation might be to change the values of global variables during a computation or to switch algorithms at runtime. When adaptivity scenarios are conceived at the start of the software design process, we can design inherently adaptive algorithms. However, when changing the behavior of an existing program, modifying the original code is often viewed as inevitable. Here, we present an alternative solution.

1521-9615/12/$31.00 © 2012 IEEE
Copublished by the IEEE CS and the AIP

Pilsung Kang, Naren Ramakrishnan, Calvin J. Ribbens, and Srinidhi Varadarajan
Virginia Tech
Michael A. Heffner
Librato

Typically, modifying a program to make it more adaptive involves changing a function call into an `if-then-else` statement that lets you choose two different functions depending on the runtime value of a predicate. In large programs with a complex adaptive plan, such rewriting can become tedious or cumbersome because you must locate and update all the places that need modification. Sometimes, such changes require restructuring the whole program, which can be quite imposing. In fact, the object-oriented (OO) programming community has identified the need to add new functionality to existing code in a modular way as *aspect-oriented programming*.[1] AOP helps abstract out a new functionality or concern for existing code into an aspect, and the aspect code can be inserted or weaved at the right places in a program.[2] This code insertion process can work even at the binary level for Java programs without having the source available. Even for non-OO languages such as C, there are tools and language extensions that enable AOP's advice-weaving constructs for code insertion.[3–4] However, we still lack comparable support for Fortran, which is widely used in scientific computing.

Binary instrumentation tools can overcome this language-dependency issue. These tools insert code to existing programs in a compiled binary form, offering clean separation between the original and the new code because the two codes are coalesced at the binary level instead of

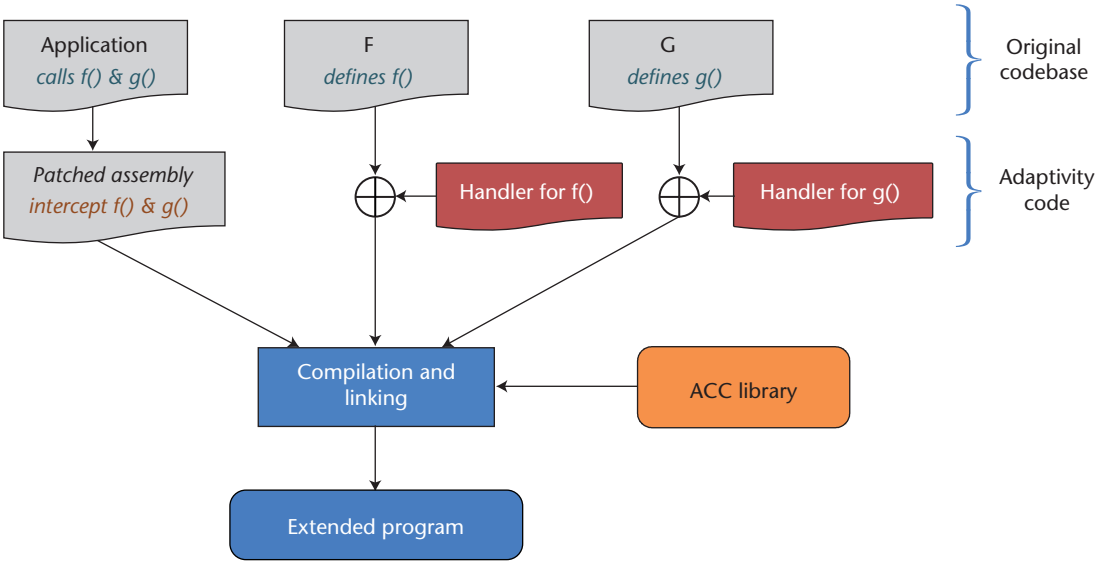

Figure 1. Adaptive application development under the Adaptive Code Collage (ACC) framework. The assembly-language version of the original source code is first patched to embed hooks at the desired function calls.

the programming-language level. Still, because they deal with the native processor instructions at the lowest level, binary instrumentation tools are typically developed for sophisticated program analysis purposes,[5-7] such as debugging and profiling, rather than as a tool to aid programmers in extending existing programs in a modular way. An exception is Detours,[8] a general-purpose package for dynamic function interception. However, Detours is available only on Windows systems.

Here, we present Adaptive Code Collage (ACC), a framework for implementing adaptive programs on top of nonadaptive existing code without resorting to AOP tools or frameworks. Instead, we use a function-call interception technique to monitor and adapt the original function behavior. As Figure 1 shows, the assembly-language version of the original source code is first patched to embed hooks at the desired function calls. Because the patching is applied at the assembly level, the original source code is unaffected and its program structure is maintained. Each of the intercepted function calls and its parameters are then associated with individual handler codes, which are separately written to perform the appropriate computations necessary to decide whether to change the function behavior. Both the patched original and the adaptivity code are then combined to produce a complete application.

Our ACC framework has three key features:

- *Modularity and transparency.* The adaptivity code is written and managed as a separate module with regard to the original program. The original code, which we assume was written in a high-level language, isn't modified. Instead, code insertion is applied to the compiler-generated assembly code.
- *Fortran support.* The assembly code patching works for Fortran as well as C. The method can be applied as long as the original code can be compiled to generate assembly output.
- *Persistency.* Code patching is needed only once. Once it's done, the patched code can be used anew without further manipulation and can be linked with any adaptivity-code variant as the program evolves. This enables the late binding of adaptivity code.

ACC's capabilities for implementing adaptivity are quite general, and in the next section we discuss the adaptive contexts when applying ACC to scientific software.

Adaptive Contexts in ACC

ACC is primarily a programmer's tool for factorizing adaptivity with an existing scientific code base to enable the plug-and-play of different adaptive strategies. It's not a specific recommendation for what needs to be adapted and how (which is the purview of domain-specific information).

Many sophisticated code bases—such as the example application in computational fluid dynamics, which we present later—have withstood years of use but are constantly put to the test as they're ported to new computational platforms

Table 1. Comparison of program modification techniques.

Features	AOP Frameworks		Adaptive Code Collage	Binary Instrumentation
	C/C++	AspectJ		
Modification level	Source language	Source or binary	Assembly	Machine instructions
Source availability	Always needed	Not necessary	Only once	Not necessary
Insertion time	Compilation time	Compilation or link time	Link time	Post-compilation
Fortran support	Bound to C/C++	Bound to Java	Yes	Yes
Main usage	Program behavior extension		Program behavior extension	Program analysis

and extended to new problem contexts and new physical modeling situations. For instance, the Generalized Incompressible Direct and Large-Eddy Simulations of Turbulence (GenIDLEST) code we describe later must be adapted for new memory hierarchies and data distributions, for modeling time-varying flow conditions, and to accommodate the latest linear system solvers. We show how the vanilla GenIDLEST code base can be adapted to these situations by factoring out the adaptivity/control decisions in a separate code and weaving them into the program execution. (Sophisticated codebases that already have adaptation built into them are outside ACC's scope.)

Although our methods can be applied to any code base, we focus on scientific codes primarily because these codes are especially rich in adaptation possibilities. In particular, they often feature libraries of multiple algorithms for the same problems, with selective superiorities between them. Secondly, adaptation is a long-used technique for improving performance and methods to automatically switch between algorithms and to pursue alternative lines of computation that are crucially important. Finally, there's a growing emphasis in scientific computing on bringing the best practices in software engineering (such as modularity and separation of concerns) into scientific code management, and ACC is a tool that works within this tradition.

A variety of adaptation strategies can be realized using ACC, such as code adaptation on a per-problem or per-execution-context basis, or even adaptation steered interactively by the user as opposed to automatically inside execution. Irrespective of the diversity of these strategies, many of them often rely on tracking and modifying a single variable or set of variables, or diverting function calls, to improve the computation's performance, stability, or accuracy. We developed ACC to support such capabilities.

Table 1 compares ACC with other program modification or extension tools. Overall, our approach is similar to the advice-weaving technique of AOP in that modular development and integration is possible. However, key points of contrast exist. Unlike AspectJ (www.eclipse.org/aspectj), for instance, ACC supports non-OO languages. Unlike AspeCt-oriented C(www.aspectc.net) or AspectC++(www.aspectc.org), which use source-to-source translation, ACC performs code modification at the assembly level.

Because ACC works at the assembly-language level, what it can and can't do must be described at a much lower granularity than high-level languages and their features. ACC makes the following assumptions in supporting an adaptivity scenario. First, the types of parameters that can be readily supported are those involving the primitive data types—such as integers, reals, and Booleans—that also include pointers. This lets us support most C/C++ codes and all Fortran codes (because Fortran uses call-by-reference to pass parameters to function calls). ACC wouldn't be directly applicable to adapting those functions that pass parameters of structured data types unless the parameters are passed by pointers.

Second, ACC's support of C++ is weak in that we can't directly apply the assembly patching for C++ methods because of C++'s name mangling, which is heavily dependent upon compiler implementations. C wrappers for C++ methods can be used to resolve the issue. Otherwise, it remains a programmer's burden to manually resolve the mismatch between different compilers. Finally, certain features that we can't support currently include dynamic functions, function parameters passed through registers, and custom orders of parameter "reading" as implemented by different compilers and languages.

Here, we present two examples of ACC use. First, we show how it adjusts the over-relaxation parameter ω in an adaptive SOR algorithm for

linear systems. This is a trivial example from textbooks, but it serves as a programmer's guide on how to use our framework. It thus enables us to delve into ACC in greater detail in our second, more complex example involving the GenIDLEST code.

As we present the details of our framework and how we instantiate it in these case studies, we make a key distinction between our framework's performance and the performance of the adapted code. The latter hinges on how cleverly relevant problem characteristics are tracked by the adaptive segments and used to improve performance. Our focus is on the former—that is, how easy ACC makes adaptation and how much overhead it introduces.

Basic Constructs for Implementing Adaptivity

A framework that can model complex adaptive scenarios must be able to compositionally switch back and forth between the old, unpatched code and the newly designed functions. We first present a set of three primitive operations that can be used as building blocks for such compositional modeling.

Function Interception

One of ACC's basic goals 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 adapted. Therefore, the framework's basic construct is a method for intercepting, or catching, the function calls made within an application.

Several programs and projects support intercepting function calls within an application.[6,8] ACC is different in that it intercepts function calls at the location the calls are made. In x86 assembly, function calls within an application are represented by the call instruction, whose single argument is the address of the function to invoke. The ACC framework targets the assembly-language representation of an application, allowing us to replace any call instruction with a call to our own interception handler. Then, at runtime, the original target function address is saved so that the interception handler can determine which function was being called. When our interception handler is called in place of the original function, the handler will determine whether to continue execution with the original target function or to perform some other operation.

When the application is modified so that function calls are intercepted by ACC, the locations at which the calls are made are modified instead of the locations associated with the target functions. This is because the code associated with the callee might not be available when the application is modified, such as when the callee is located in a dynamic module that isn't automatically loaded at runtime. By modifying the caller instead, we don't require all the components of an application to be loaded at runtime. We convert the original call instructions—which would otherwise require correct runtime link binding—into lookup references that ACC uses to determine the appropriate action to take when the calls are made. The original function-call location in effect becomes a placeholder (or "link point," in the AOP vernacular) that the framework can connect to arbitrary procedures during runtime.

To allow for the most adaptivity in a procedure-level decomposition, the framework must also support the ability to catch when a function returns. In ACC, when a function is diverted through our interception handler, we manipulate the return address to point to a return-handler instead. The return-handler can perform post-function-call computation—after which it will eventually return to the original return address. This is useful because when the function returns, the function call's outcome can be queried, including the return value or any values returned via pass-by-reference parameters. For example, this allows the adaptivity code to massage any return values before they're passed back to the calling module to fix type differences or influence the caller.

Registered Callbacks

When a function call is intercepted or the return of a function is intercepted, the framework passes control of the application to an adaptation module so that it can make any required adaptive control decisions. This functionality is supported in ACC using a method of registered callbacks.

An adaptivity module can register either a pre- or post-callback (or both) for a given function, which is executed when the function is invoked or a return is made from the function, respectively. When ACC invokes a callback, it passes a reference to the invocation stack entry, an ACC data structure for bookkeeping function-invocation information corresponding to that function call. With the invocation entry reference, the callback can lookup or manipulate parameters, remap the function call, and search the remaining invocation stack. In other words, the callback has full control over the application's current state.

Function-Call Parameter Manipulation

We provide a complete set of controls that let the application 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 sufficient progress or whether any adaptive decisions should be made to improve the results or performance. For example, a recursive sorting routine's subinterval indices can reveal whether it might be beneficial to switch to a different sorting algorithm. Similarly, tracking a numerical routine's current relative error might reveal that switching to a different routine would improve convergence.

ACC supports the ability to query and manipulate a function's parameters before and after its lifetime. Before any operations on the parameters can be performed, the size and type of each parameter must be specified so that the framework can calculate each parameter's correct memory offset. Or the predefined parameter types that represent which type and size can be used. Once all function parameters are specified to ACC, simple query functions will return pointers to the parameters in memory and, by dereferencing these pointers, the application can read/write the parameters in memory. Also, parameters that are passed by reference to a function can be manipulated when a function returns if the user desires to massage the values returned.

However, the most useful parameter construct of our ACC framework is the ability to remap a function's entire parameter list. For example, when an adaptation scheme intends to remap one function to another, the function signatures typically must be the same because the parameters are already on the stack. This severely limits the flexibility to abstractly connect system components if you don't know the correct semantics beforehand. If the adaptation code specifies the parameters' size and type of both the original function and the new function, you can overcome this barrier by simply

- calculating the difference in lengths of the two parameter lists,
- adjusting the frame pointer by the difference in lengths, and
- copying the new parameters onto the stack.

Thus, ACC's function call parameter manipulation functionality allows for realizing complex adaptation scenarios.

Application Development Process

The adaptive application development process under ACC consists of three stages: patching the compiler-generated assembly language from the original code, writing the code for an adaptation scenario, and writing the glue code that connects the adaptivity component to the original code through the ACC APIs.

Patching the Original Code

Our approach is to perform the necessary code modification over the GCC assembler output generated from the source before it's assembled to an object file. Applying a simple pattern-matching and substitution Perl script over the assembly code, we replace every function call of interest with a call to the framework's `acc_func_intercept` function, a general hook for branching to individual handler code to adapt the original behavior of functions. As explained earlier, we apply the patch to the program code that calls the target function, not to the function code itself.

Because the code patching is done at the assembly level, the adaptive application development process is cleanly separated from the development activity of the original program. As long as the original program is written in high-level languages such as Fortran or C, the source and its program structure are unaffected.

Another advantage of our code-patching technique is the persistence of the hooks inserted into the original program. Once the code is patched and compiled into an object file, because the hooks are now in there, it can be linked with different versions of the handler code without recompiling the original application code as the handler code evolves.

Writing Handler Code for an Adaptation Scenario

After the original code is patched to intercept specific function calls, you must write the handler code to implement any adaptive operations that will be performed at the trapped calls; you can choose to trap before and/or after the execution of the function call. The operations in the handler code will be uniquely determined depending on the associated original function and the given application. Any handler code, however, must do two things: check the current program state as seen at the time of function call and change the function behavior if needed. To support these operations at the level of functions, our API provides the following functions:

```
/* return a pointer to the pos'th
   parameter on the function call
   stack */
void *acc_get_param (struct acc_invoke_
   entry *ie, int pos)

/* remap from the current trapped
   function to a new function, remap_
   func */
void acc_remap (struct acc_invoke_
   entry *ie, acc_invoke_func_t
   *remap_func)
```

When a function call is trapped, `acc_func_intercept` passes to the associated handler an instance of `struct acc_invoke_entry`, which is a data structure that keeps track of an intercepted function and stores its accompanying information (such as the address of the original arguments). Then `acc_get_param` takes the `acc_invoke_entry` instance and returns a pointer to *pos*th input parameter to the original function, letting you access and modify its value. Macros for different datatypes are also provided for easy access. For example, `ACC_GET_PARAM DOUBLE(ie, pos)` returns the value of the *pos*th parameter, which is known to be of type `double`.

Other than dealing with input parameters, the function behavior can be completely redefined through the use of `acc_remap`, which replaces the current call to the original function with a call to a new function, `remap_func`. ACC uses the `acc_invoke_func_t` type for generic functions.

Although a function's behavior is changed by intercepting function calls, global variables that aren't communicated through function parameters can be accessed by declaring them as external variables. For instance, the variables in a Fortran common block can be accessed by defining a `C extern struct` global variable that matches the common block. Or, in cases where only a few of the variables in a common block are needed, simple `getter` and `setter` Fortran subroutines with the same common block can be employed to read and write the variables.

Writing the Glue Code

The glue code performs any initialization necessary to combine the patched original code with its handler code into a complete adaptive application. The functions to be intercepted are added to the symbol table managed by ACC, and their corresponding handler functions are associated through the registration API. In addition, to retrieve the intercepted function parameters from the stack, the type information of each parameter must be specified so that ACC can determine the correct memory offsets for the parameters.

```
ACC_PARAM_TYPE params[3]; /* 3 parameters for sort */
struct acc_symbol_entry *se; /* symbol entry for a
   function */

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

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

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

/* register a pre-handler for sort */
acc_add_handler(ACC_HT_PRE, "sort", sort_handler);
```

Figure 2. The initialization for dynamically intercepting the `sort` calls and accessing its parameters using the ACCAPIs. A symbol entry and a parameter list for `sort` are declared and associated together, and a handler for `sort`—`sort_handler`—is registered to ACC's runtime.

These initialization steps can be performed in the glue code's `main` function if the original code doesn't have the C `main` entry, which is typical for software library packages or Fortran applications that use `MAIN_` as its starting entry when compiled with GNU or Intel compilers. For C applications with a `main`, this is the one place where the source code must be modified to include the initialization steps.

Consider a simple function called `sort`,

```
void sort (int *data, int first, int
   last),
```

where `data` is the input array of unsorted integers, and `first/last` is the index of the first and last elements, respectively, which signifies the local subarray to sort. The code in Figure 2 shows the initialization for dynamically intercepting the `sort` calls and accessing its parameters using the ACCAPIs.

`ACC_PARAM_TYPE` is an enumerated type that abstracts a set of different data types of C function parameters. For instance, `ACC_PARAM_POINTER` and `ACC_PARAM_INT` are used for C pointers and integers, respectively, in Figure 2's code.

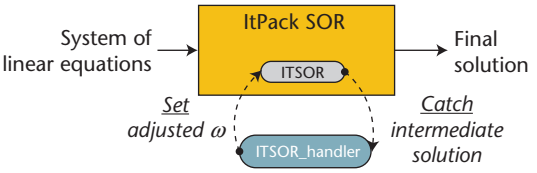

Figure 3. Adaptive Successive Over-Relaxation (SOR) system through interception of `ITSOR` calls using ACC. Because ACC acquires program control at each `ITSOR` call, it augments the desired adaptivity computation externally and transparently to both the ItPack library and the user's code.

As described earlier, only primitive datatypes are currently supported by ACC.

After each parameter's data type is assigned, `acc_add_params_list` associates the parameter list with the `sort` function's symbol entry, which has been previously constructed through the `acc_add_symbol` function:

```
struct acc_symbol_entry *acc_add_
  symbol(const char *name, acc_invoke_
  func_t *func)
```

For a given function `func`, it takes an additional parameter to construct a symbol: `name` for a user-defined symbol name. Finally, we use the `acc_add_handler` API to register `sort_handler` as a callback for `sort` that is identified by the "sort" symbol. This is done with `ACC_HT_PRE` to transfer the execution control to `sort_handler` before the call to `sort` executes, which is similar to the `before` advice in AOP. Likewise, `ACC_HT_POST` can be used to operate like the `after` advice in AOP. You can also implement a desired adaptation logic in the body of the `sort_handler` function for adapting the original `sort` function.

Case Study 1: Adaptive Successive Over-Relaxation

We applied ACC to the Successive Over-Relaxation (SOR) routine found in ItPack 2C,[9] a package of Fortran subroutines for solving linear systems by adaptive iterative methods. Like many iterative solvers, the convergence rate of SOR is heavily influenced by a parameter—in this case, the over-relaxation parameter ω.

ItPack includes an internal algorithm for automatically adapting ω. However, this algorithm is based on a simple heuristic that works reasonably well for a wide range of matrices, but can't take advantage of problem-specific information that might be available in a given instance.

With ACC, we can externally adapt the algorithm by controlling the choice of ω without modifying user code or the ItPack library. Here, we illustrate this use of our framework, comparing a new approach to adapting ω using ACC against ItPack's own version of adaptivity. The purpose of this example isn't to propose a new adaptive SOR algorithm, but rather to show ACC's applicability to changing the behavior of scientific code using an external model. This is similar to work by other researchers,[10] which applies automatic differentiation to a Fortran SOR code to adjust the ω parameter.

Implementation

Each iteration of the SOR algorithm in ItPack is performed by the `ITSOR` subroutine. Therefore, we patch the SOR code to intercept the `ITSOR` calls, thereby transferring program control to ACC (see Figure 3). In this way, we can access and examine the whole set of subroutine parameters, including the solution vector, at the time of the call. More importantly, because ACC acquires program control at each `ITSOR` call, we can augment the desired adaptivity computation externally and transparently to both the ItPack library and the user's code. The patched SOR code is then assembled and linked with the rest of the ItPack code to build a complete object module, thereby enabling interception of the `ITSOR` calls within the SOR routine.

To access ω, which is internal to `ITSOR` and not communicated through any of the subroutine arguments, we supply simple getter and setter routines where we declare ItPack's common block `ITCOM3` to which the parameter belongs. The getter routine checks the value of ω at the current step and the setter updates ω with a new value for the next iteration.

We apply our adative SOR code to the 2D Poisson problem

$$\nabla^2 u(x,y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - f(x,y)$$

posed on the unit square with Dirichlet boundary conditions. Applying an $N \times N$ finite difference mesh and standard centered finite diferences, the SOR iteration is given by

$$u^{GS}_{(i,j)} = (h^2 f_{(i,j)} + u_{k+1,(i-1,j)} + u_{k,(i+1,j)} + u_{k+1(i,j-1)} + u_{k,(i,j+1)}) / 4,$$
$$u_{k+1,(i,j)} = (1-\omega)u_{k,(i,j)} + \omega u^{GS}_{(i,j)},$$

where $1 \le i, j \le N$, $h = 1/(N+1)$, and $u_{k,(i,j)}$ is the element of the solution vector u_k corresponding to the mesh point (i, j) at iteration k. The residual

Table 2. Comparison of adaptive SOR algorithms for Poisson problems.

Simulation Parameters			Fixed $\omega = \omega_{opt}$		ItPack		Our Method		Speedup Time
N	ω_{opt}	ζ	N_{iter}	Time(sec)	N_{iter}	Time(sec)	N_{iter}	Time(sec)	(%)
300	1.979	10^{-2}	283	0.73	601	1.52	438	1.12	36
400	1.984		385	1.80	800	3.68	587	2.73	35
500	1.987		493	3.62	1,087	7.88	802	5.92	34
750	1.992		866	14.2	1,579	25.7	1,268	20.9	23
1,000	1.994		1189	34.4.	2,222	63.1	1,884	53.0	19
300	1.979	10^{-3}	422	1.07	735	1.85	630	1.59	16
400	1.984		568	2.66	965	4.45	718	3.34	33
500	1.987		717	5.33	1,279	9.30	988	7.24	28
750	1.992		1172	19.2	1,855	30.3	1,549	25.4	19
1,000	1.994		1590	46.4	2,848	82.1	2,281	66.1	24
300	1.979	10^{-4}	555	1.39	859	2.19	728	1.95	12
400	1.984		746	3.45	1,117	5.16	813	4.10	26
500	1.987		943	6.94	1,495	10.88	1,268	9.23	18
750	1.992		1503	24.6	2,193	36.0	1,788	30.2	19
1,000	1.994		2003	58.1	3,599	103.1	2,694	77.0	34

N = problem size; N_{iter} = number of iterations; ω_{opt} = optimized over-relaxation parameter; and ζ = the error-tolerance parameter.

norm for the Poison problem at the kth step is given by

$$r_k = \| b - Au_k \|,$$

where b is the right-hand side vector and A is the discrete Laplacian.

Two questions must be addressed in implementing adaptive SOR: when to change ω, and how to obtain its new value. To decide the right time to change ω, we simply borrow ItPack's scheme based on convergence rate estimation[11] and implement it in our handler code. Because the purpose of this example is to illustrate the applicability of ACC in composing an adaptive application, rather than to devise a new adaptive algorithm, adopting an already established schemesuits our needs well. Once we decide to change ω, we fetch the intermediate solution vector fromthe parameter list of `ITSOR` and calculate the residual norm r_k, which is then used in a simple method to choose the new value of ω:

$$\omega_{k+1} = \frac{1}{2}\left(\omega_{k+1}^{(1)} + \omega_{k+1}^{(2)}\right),$$

where we view the residual r_k as a function of ω, so that $\omega_{k+1}^{(1)}$ is the next value given by a secantiteration seeking to solve $r_{k+1}(\omega) = 0$, and $\omega_{k+1}^{(2)}$ is the next value given by a secant iteration seeking to minimize $r_{k+1}(\omega)$—by looking for a root of an approximation to $r'_{k+1}(\omega)$. Because both estimates $\omega_{k+1}^{(1)}$ and $\omega_{k+1}^{(2)}$ require values from two previous iterations, we bootstrap the process by using the default ItPack scheme for computing a new ω the first time ω is adjusted.

Experimental Results

We use $f(x, y) = 0$ and constant Dirichlet boundary conditions for the Poisson problem. We applythe ItPack SOR solver to the discretized problem with our adaptivity framework turned on andthe internal adaptivity capability of ItPack turned off. The iteration terminates according to the `ITSOR` stopping test.[11]

On a 32-bit, x86 Linux machine running Fedora Core 6 with an Intel Pentium D dual-core 3.60-GHz CPU and 2 Gbytes of RAM, we compare the performance of our version of adaptive SOR with that of ItPack, both of which are compiled with GNU Fortran 95 (gfortran) 4.1.1 with O3 optimization turned on. We also measure the case where ω is fixed at the optimal value; the optimal value of ω for an $N \times N$ Poisson problem is

$$\omega_{\text{opt}} = \frac{2}{1 + \sin\left(\frac{\pi}{N+1}\right)}.$$

Table 2 shows both the execution time (average of three runs) and the number of iterations required

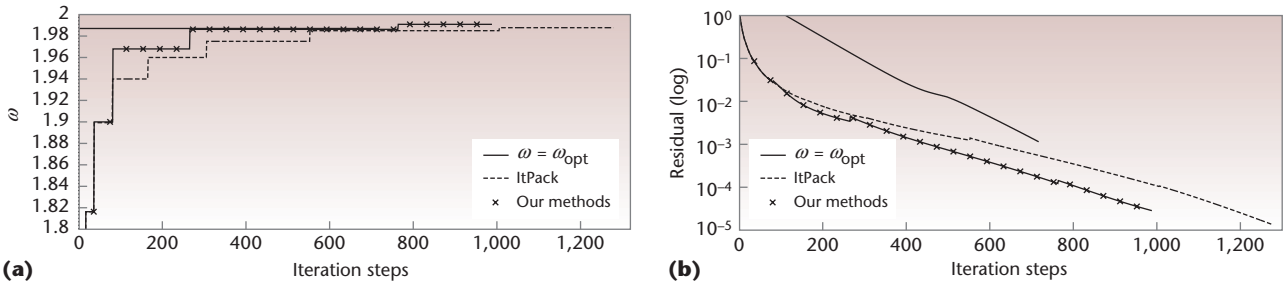

Figure 4. Comparison of SOR methods for a 500 × 500 Poisson problem. (a) The adaptive progress of ω over iterations (b) The residuals over iterations reduces as the computations proceed.

to converge for different problem sizes ranging from $N = 300$ to $N = 1{,}000$, along with ζ (the error-tolerance parameter), ranging from 10^{-2} to 10^{-4}. The value of ω was initially set to 1.5 for both adaptive SOR programs. As the table shows, our adaptive SOR performed better than ItPack's in all cases, with speedup as high as 36 percent in terms of execution time.

In addition, using the ACC framework doesn't entail significant overhead. For instance, intercepting `ITSOR` function calls for a 300 × 300 problem with ζ set to 10^{-2}—the smallest problem, where the framework overhead would be the most obvious—increased execution time by only 0.01 seconds on average (that is, less than 0.7 percent).

Figure 4a shows the adaptive progress of ω over iterations for a 500 × 500 Poisson problem with ζ set to 10^{-3}, starting from $\omega = 1.5$. In all, ω is adapted seven times in this example, but we show only the last five for two reasons:

- to highlight the differences between our method and the default ItPack scheme, and
- because our first two adaptive steps actually use the ItPack scheme, as our estimator needs two initial points.

As the figure shows, our adaptivity method adjusts ω more quickly toward ω_{opt} than ItPack, although it overestimates a bit at the end. The execution time improves by 28 percent, but the handler function for `ITSOR` consumes only 0.044 seconds on average during the whole execution, which amounts to 0.6 percent of the total 7.24 seconds. Figure 4b shows the reduction in residual as the computations proceed. (The residuals were computed explicitly at each iteration in a separate run.) As the figure shows, our adaptive approach succeeds in improving the rate of residual reduction compared to ItPack's adaptive scheme. ItPack uses a relative error estimate, rather than a residual estimate, to terminate the iterative process. This explains why the ω_{opt} case terminates first, despite having a larger residual for a given iteration than the other two cases, both of which focus on minimizing the residual when adapting ω.

Case Study 2: Parallel CFD Codes

We used the ACC framework to implement various adaptivity scenarios in the context of real scientific computing codes, including applications in biochemical network simulation[12] and computational fluid dynamics (CFD).[13,14]

As we describe elsewhere,[13] ACC is used to improve the performance, stability, and accuracy of GenIDLEST, a large parallel CFD code.[15] GenIDLEST is written in Fortran 90 with a message passing interface (MPI), and solves the time-dependent incompressible Navier-Stokes and energy or temperature equations. The GenIDLEST time integrator's stability depends on the timestep used, but it's difficult to identify a single adaptation scheme that will automatically and successfully adjust the timestep forthe wide variety of problems that engineers use GenIDLEST to solve. Hence, in practice, users save checkpointed solutions periodically during the long simulation runs, so that if instability occurs they can restart the simulation from the last stable state with a smaller timestep. By using ACC to plug in a separately written stability module, we factor out the time-step adjusting strategy from the main code base, allowing users to easily use and experiment with different strategies, as appropriate.

In other work,[13] we show how one particular set of CFD computations can be stabilized effectively by a simple multiplicative increase/decrease algorithm, where the time step is increased (to reduce time-to-solution) or decreased (to maintain numerical stability) by a preset factor if the computed Courant-Friedrich-Levi (CFL) indicator goes above or below preset upper and lower CFL thresholds.

Figure 5 shows the results for GenIDLEST enhanced with the adaptivity module for a typical simulation, with different initial values of time

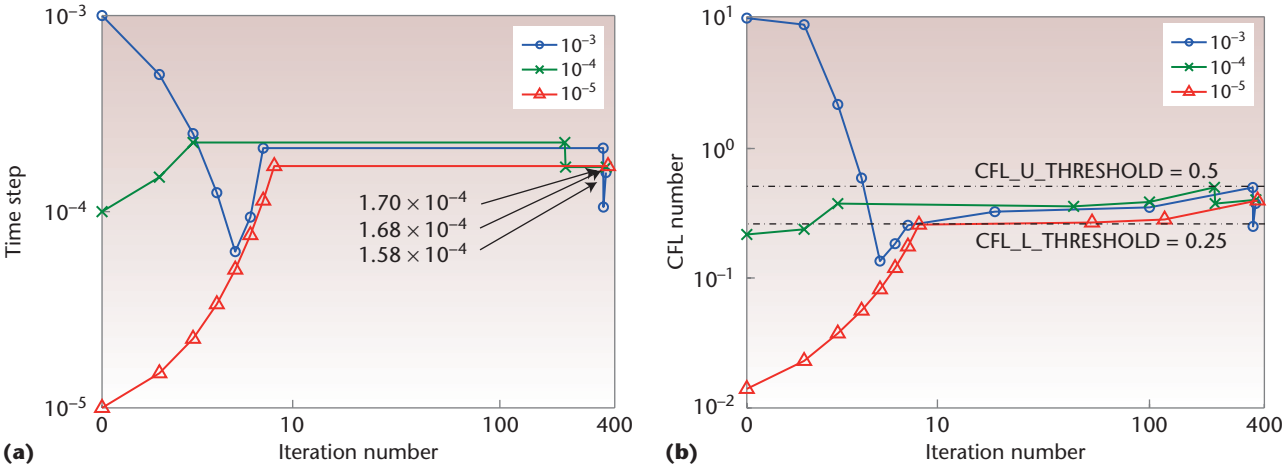

Figure 5. Automatic adjustment of the time-step parameter in GenIDLEST.[13] (a) Time-step change and (b) the Courant-Friedrich-Levi (CFL) number.

step ranging from 10^{-3} to 10^{-5}. The CFL lower and upper thresholds were set to 0.25 and 0.50, respectively. The graphs show how the CFL value changes as the time-step parameter is controlled by the new module, thereby maintaining the simulation's stability.

In addition to automatic time-step adaptation, we also use ACC to facilitate an interactive, user-controlled adaptation of GenIDLEST's flow model, resulting in more accurate simulations. In CFD simulations, the predicted flow characteristics depend on the selection of the appropriate flow model. In problems of interest, it's critical to choose an appropriate model when the simulated flow is in the transition region between laminar and turbulent. Our ACC-enabled mechanism lets users monitor the stream-wise velocity's time variation, which can be used to infer when to switch from a laminar flow model to a turbulent model, and when to switch from one turbulent flow model to a more accurate but computationally expensive alternative. Because GenIDLEST already implements all the flow models, with a particular one selected by setting a single parameter, it's easy to use ACC to change models by controlling this parameter, which requires no modification to the original code.

We're also using ACC to support dynamic methods for performance tuning of algorithmic parameters in parallel scientific codes. An important trend in high-performance scientific computing is the use of auto-tuned or self-adaptively optimized algorithms and implementations. Approaches include language extensions,[16] model-driven compiler optimizations,[17] and empirical search-based schemes.[18] Although these methods have been successful in limited domains (such as numerical linear algebra kernels), there's still a need for better support for application-specific adaption schemes, where a particular computation's unique context—including the code, the data, and the computing resources—can be taken into account.

In earlier work, we showed how to use ACC to implement a dynamic method for tuning algorithmic parameters in codes such as GenIDLEST.[14] For example, we inserted adaptive schemes to adjust two parameters that strongly influence the performance of the preconditioner used in an important linear solution step. The first parameter is the size of a subdomain block, represented by nb_i, nb_j, and nb_k. This parameter defines the structure of the domain decomposition preconditioner; it influences, often in nonobvious ways, both the preconditioner's quality as an approximation to the original discrete PDE operator and the computation's floating-point performance (such as through memory hierarchy effects). The second parameter is ns, the number of inner relaxation sweeps used in the multilevel preconditioner.

Figure 6 shows the performance of GenIDLEST for a typical problem for 3,000 time steps, where each point corresponds to the elapsed time measured every 50 steps during the simulation. The thick gray solid line corresponds to the tuned algorithm, and the colored lines show the performance for other typical fixed choices of the parameters. The tuned curve is labeled at various points to show the history of the simple adaption scheme used, where the parameters nb_i, nb_j, nb_k, and ns are adjusted automatically to test and improve performance as the simulation proceeds. Overall, the time-to-solution for the full 10,000 time-step run improved by 26 percent over the performance of the nonadapted code, with typical parameter choices.

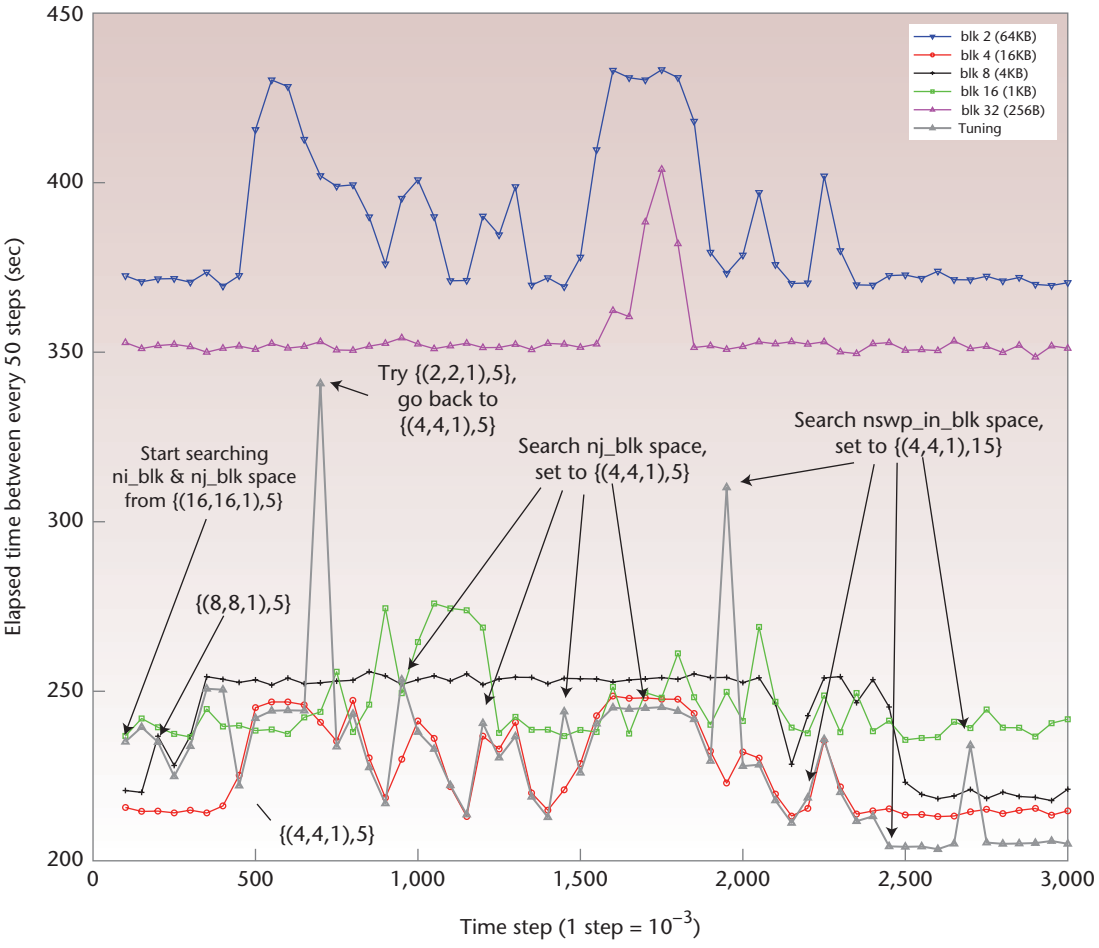

Figure 6. Dynamic tuning of preconditioning parameters in the Generalized Incompressible Direct and Large-Eddy Simulations of Turbulence (GenIDLEST).[1] This problem included 3,000 time steps, where each point corresponds to the elapsed time measured every 50 steps.

Modern numerical methods often have several such parameters, and their influence on accuracy and performance is hard to predict for realistic problems running on a particular computational resource. Dynamic tuning is often the only effective approach. With ACC, we can factor out this important concern from the standard code base, allowing much greater flexibility in deciding what and how to adapt.

We are applying ACC to more diverse areas in scientific computing. In particular, we are exploring more dynamic scenarios of scientific software adaptation, where the user's runtime decisions can be supported to realize flexible simulations, without complete description of adaptation specifications even before application launch. We are also investigating recurring adaptation scenarios in scientific computing that can be abstracted out into pattern templates, so that they can be easily reused for different application domains in a more manageable way.

ACC is available for download at http://people.cs.vt.edu/~kangp/ack. CiSE

References

1. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming*, vol. 1241, Springer-Verlag, 1997, pp. 220–242.
2. E. Hilsdale and J. Hugunin, "Advice Weaving in AspectJ," *Proc. 3rd Int'l Conf. Aspect-Oriented Software Development*, ACM Press, 2004, pp. 26–35.
3. O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language," *Proc. 40th Int'l Conf. Tools Pacific*, Australian Computer Society, 2002, pp. 53–60.
4. W.R. Mahoney and W.L. Sousan, "Using Common Off-the-Shelf Tools to Implement Dynamic Aspects," *Sigplan Notes*, vol. 42, no. 2, 2007, pp. 34–41.

5. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. ACM Sigplan 1994 Conf. Programming Language Design and Implementation*, ACM Press, 1994, pp. 196–205.
6. B. Buck and J.K. Hollingsworth, "An API for Runtime Code Patching," *Int'l J. High Performance Computing Applications*, vol. 14, no. 4, 2000, pp. 317–329.
7. C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, "*Proc. ACM Sigplan Conf. Programming Language Design and Implementation*, ACM Press, 2005, pp. 190–200.
8. G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," *Proc. 3rd Usenix Windows NT Symp.*, Usenix Assoc., 1999, pp. 135–144.
9. D.R. Kincaid et al., "ITPACK 2C: A Fortran Package for Solving Large Sparse Linear Systems by Adaptive Accelerated Iterative Methods,"*ACM Trans. Mathematical Software*, vol. 8, no. 3, 1982, pp. 302–322.
10. P.D. Hovland and M.T. Heath, *Adaptive SOR: A Case Study in Automatic Differentiation of Algorithm Parameters*, tech. report ANL/MCS-P673-0797, Mathematics and Computer Science Division, Argonne Nat'l Lab., 1997.
11. L.A. Hageman and D.M. Young, eds., "The Successive Overrelaxation Method," *Applied Iterative Methods*, Academic Press, 1981, pp. 223–233.
12. P. Kang et al., "Modular Implementation of Adaptive Decisions in Stochastic Simulations," *Proc. 24th ACM Symp. Applied Computing*, ACM Press, 2009, pp. 995–1001.
13. P. Kang et al., "Modular, Fine-Grained Adaptation of Parallel Programs," *Proc. 9th Int'l Conf. Computational Science*, Springer Verlag, 2009, pp. 269–279.
14. P. Kang et al., "Dynamic Tuning of Algorithmic Parameters of Parallel Scientific Codes," *Proc. 10th Int'l Conf. Computational Science*, Springer Verlag, 2010, pp.145–153.
15. D. Tafti, "GenIDLEST—A Scalable Parallel Computational Tool for Simulating ComplexTurbulent Flows," *Proc. ASME Fluids Eng. Division* (FED), vol. 256, Am. Soc. Mechanical Engineers, 2001, pp. 347–356.
16. C.A. Schaefer, V. Pankratius, and W.F. Tichy, "Atune-IL: An Instrumentation Languagefor Auto-Tuning Parallel Applications," *Proc. 15th Int'l Euro-Par Conf. Parallel Processing*, Springer Verlag, 2009, pp. 9–20.
17. K. Kennedy and J.R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*, Morgan Kaufmann, 2002.
18. J. Demmel et al., "Self-Adapting Linear Algebra Algorithms and Software," *Proc. IEEE*, vol. 93, no. 2, 2005, pp. 293–312.

Pilsung Kang *is a senior engineer at Samsung Electronics, where he develops embedded software for solid-state drives. His research interests include computational science, software engineering, parallel computing, and embedded systems. Kang has a PhD in computer science from Virginia Tech. Contact him at kangp@cs.vt.edu.*

Michael A. Heffner *is a lead developer at Librato, where he works on the Silverline application-monitoring and management solution. His research interests include large-scale, distributed architectures, high-performance computing, and systems development. Heffner has an MS in computer science from Virginia Tech. Contact him at mike.heffner@librato.com.*

Naren Ramakrishnan *is a professor and associate head for graduate studies in the Department of Computer Science at Virginia Tech. His research interests are mining scientific data, computational science, and information personalization. Ramakrishnan has a PhD in computer science from Purdue University. He is an ACM Distinguished Scientist and serves on the editorial boards of several journals, including* Computer *and* Data Mining and Knowledge Discovery. *Contact him at naren@cs.vt.edu.*

Calvin J. Ribbens *is an associate professor and the associate department head for undergraduate studies in the Department of Computer Science at Virginia Tech. His research interests include parallel computation, numerical algorithms, mathematical software, and tools and environments for high-performance computing. Ribbens has a PhD in computer science from Purdue University. Contact him at ribbens@cs.vt.edu.*

Srinidhi Varadarajan *is the director of the Center for High-End Computing Systems and an associate professor in the Department of Computer Science at Virginia Tech. His research interests are in the area of high-end computing systems, focused more specifically on fault tolerance in large-scale distributed systems, runtime systems, and frameworks for integrated emulation and simulation of computer networks. Varadarajan has a PhD in computer science from Stony Brook University. Contact him at srinidhi@cs.vt.edu.*

Selected articles and columns from IEEE Computer Society publications are also available for free at http://ComputingNow.computer.org.