Adaptive Code Collage:
A Framework to Transparently Modify Scientific Codes

Pilsung Kang[1], Michael A. Heffner[2], Naren Ramakrishnan[1],
Calvin J. Ribbens[1], and Srinidhi Varadarajan[1]

[1]*Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, USA*
[2]*Librato Inc., Blacksburg, VA 24060, USA*

Abstract

Legacy scientific codes are often re-purposed to fit adaptive needs, such as to dynamically alter parameters to improve convergence behavior, or to switch algorithms at runtime for greater accuracy of modeling. Given a legacy scientific code, how can we make it adaptive without making changes to the original source program(s)? We present an approach—Adaptive Code Collage (ACC)—to achieve this goal using function call interception in a language-neutral way at link time. ACC transparently 'catches' function calls and redirects them so that an existing program can be made adaptive without causing a significant performance overhead. We demonstrate the appliction of ACC to designing adaptive SOR algorithms for solving linear systems and to improving the performance, stability, and accuracy of GenIDLEST, a large parallel computational fluid dynamics code.

Keywords: Scientific software adaptation, program modification, function call interception

Downloadable at: http://people.cs.vt.edu/∼kangp/ack.

1 Introduction

Adaptive programs are programs that can change their runtime behavior to achieve certain purposes (e.g., performance improvement, better numerical stability) by tracking changes in problem characteristics, available resources, and execution environments. Almost every computational scientist has encountered the situation wherein they would like to make a legacy code adaptive to their application context but without making serious changes to the original program(s). (Changing values of global variables during a computation and switching algorithms at runtime are simple examples of adaptation needs.) When the adaptivity scenario can be conceived from the start of the software design process, we can design inherently adaptive algorithms. However, to change the behavior of an already existing program, modification of the original code is often viewed as inevitable. We present an alternative solution here.

Typical modifications to make a program adaptive involve changing a function call into an if-then-else statement where two different functions may be selectively chosen depending on the runtime value of a predicate. Such rewriting may become tedious or cumbersome in large programs with a complex adaptive plan, because the programmer has to locate and update all the places where the modification is needed. Sometimes such changes require restructuring of the whole program, which can be quite imposing. In fact, this issue of adding new functionality to existing code

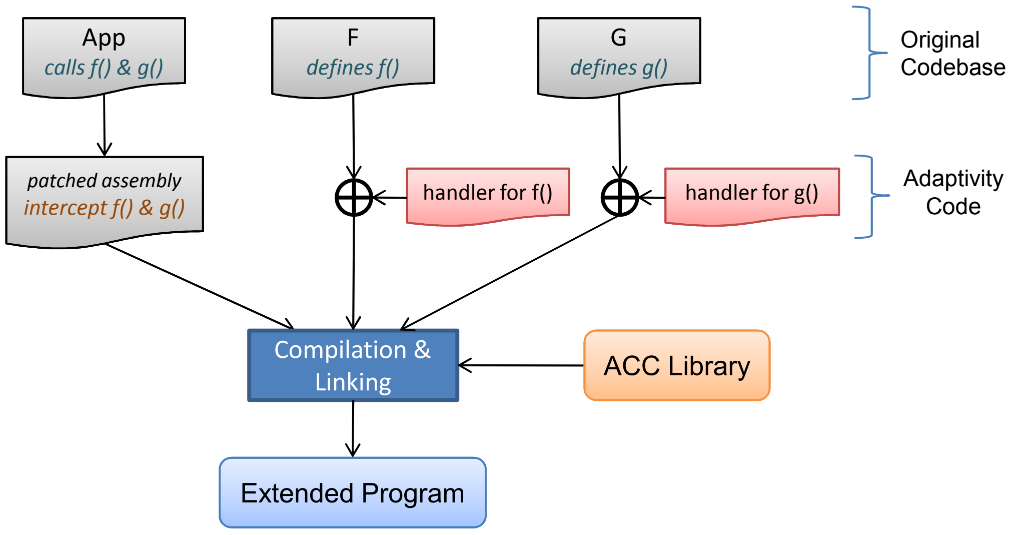


Figure 1: Adaptive application development under the ACC framework

in a modular way has been identified in the OOP (Object-Oriented Programming) community as AOP (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 like C, there are tools and language extensions that enable AOP's *advice weaving* constructs for code insertion [3–5]. However, comparable support for Fortran, which is widely used in scientific computing, is still lacking.

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 the programming language level. Still, since they deal with the native processor instructions at the very lowest level, most of those tools are developed for sophisticated program analysis purposes [6–9] 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 [10], a general-purpose package for dynamic function interception. However, Detours is only available on Windows systems.

In this paper, we present Adaptive Code Collage (ACC), a framework for implementing adaptive programs on top of non-adaptive 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 shown in Figure 1, the assembly language version of the original source code is first patched in order to embed hooks at the desired function calls. Since the patching is applied at the assembly level, the original source code is not affected 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 together to produce a complete application.

The important features of our ACC framework are as follows:

- Modularity & Transparency: The adaptivity code is written and managed as a separate module with regard to the original program. The original code, assumed to have been written

	AOP frameworks		**ACC**	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 time
FORTRAN SUPPORT	bound to C/C++	bound to Java	**yes**	yes
MAIN USAGE	program behavior extension		**program behavior extension**	program analysis

Table 1: Comparison of program modification techniques

in a high level language, is not 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 is done, the patched code can be used anew without further manipulation and can be linked with any variants of adaptivity code as the program evolves. This enables the *late binding* of adaptivity code.

2 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 is *not* a specific recommendation for what needs to be adapted and how (which is the purview of domain-specific information).

Many sophisticated code bases (see an example application to computational fluid dynamics presented later) have withstood years of use but are being constantly put to the test by being ported to new computational platforms, and by requiring their extension to new problem contexts and new physical modeling situations. For instance, the GenIDLEST code described later needs to 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 made to adapt to these situations by factoring out the adaptivity/control decisions in a separate code and weaving them into the program execution. Conversely, sophisticated code bases that already have adaptation built into them are outside the scope of ACC.

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 and there are selective superiorities between these algorithms. Secondly, adaptation is a long-used technique for improving performance and methods to automatically switch between algorithms and to pursue alternative lines of computation are very important. Finally, there is a growing emphasis in scientific computing of bringing best practices in software engineering (modularity, separation of concerns, to name a few) into scientific code management, and ACC is a further tool in this tradition.

A variety of adaptation strategies can be realized using ACC: code adaptation on a per-problem basis, on a 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/modifying a single/set of variables, or diverting function calls, in order to improve performance, stability, or accuracy of the computation. It is toward supporting these capabilities that ACC is targeted.

Table 1 compares ACC with other program modification or extension tools. Overall, our approach is very similar to the advice weaving technique of AOP in that modular development and integration is possible. However, key points of contrast can be made. Unlike AspectJ[1], for instance, ACC supports non-OO languages. Unlike AspeCt-oriented C[2] or AspectC++[3], where source-to-source translation is used, code modification is performed at the assembly level in ACC.

Since ACC works at the assembly language level, what it can and cannot do needs to be described at a much lower granularity than that of 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, boolean, etc. and also include pointers. This enables us to support most C/C++ codes and all Fortran codes (since Fortran uses call-by-reference to pass parameters to function calls). ACC would not 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 cannot directly apply the assembly patching for C++ methods due to 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 compiler. Finally, certain features that we cannot support currently include dynamic functions, function parameters passed through registers, and custom orders of parameter 'reading' as implemented by different compilers and languages.

We present two illustrations of how ACC is used: i) an example of adjusting the over-relaxation parameter ω in an adaptive SOR algorithm for linear systems, ii) a more complex example involving the GenIDLEST code. The former is a trivial example from textbooks but serves as a programmer's guide on how to use our framework and, hence, we delve into it in greater detail.

We now present the details of our framework and how we instantiate it toward the above case studies. In the following, it is important to distinguish the performance of our framework from 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, namely how easy does ACC make adaptation, and how much overhead it introduces.

3 Basic Constructs for Implementing Adaptivity

To realize our goal of a framework that can model complex adaptive scenarios, a key requirement is to 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.

3.1 Function Interception

One of the basic goals of ACC 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

[1]http://www.eclipse.org/aspectj

[2]http://www.aspectc.net

[3]http://www.aspectc.org

execution can be adapted. Therefore, the basic construct required in the framework is a method for intercepting, or catching, the function calls made within an application. There are programs and projects that support intercepting function calls within an application [7,10]. ACC is different from these projects 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 at the time the application is modified. The callee might be located in a dynamic module that is not automatically loaded at runtime. By modifying the caller instead, we do not require all 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 what appropriate action to take when the calls are made. The original function call location in effect becomes a place-holder (or "link point," in the vernacular of the AOP literature) that the framework can connect to arbitrary procedures during runtime.

In order 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 outcome of the function call 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 are passed back to the calling module to fix type differences or influence the caller.

3.2 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 may 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 whenever 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, which corresponds 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 current state of the application.

3.3 Function Call Parameter Manipulation

We provide 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 run-time 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, the subinterval indices of a recursive sorting routine can reveal whether it might be beneficial to switch to a different sorting algorithm. Similarly, tracking the current relative error of a numerical routine might reveal characteristics of the problem that suggest switching to a different routine to improve convergence.

ACC supports the ability to query and manipulate the parameters to a function 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 the correct memory offset of each parameter, or the predefined parameter types which represent both type and size can be used. Once all the parameters for a function 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 the entire parameter list of a function. For example, when an adaptation scheme intends to remap one function to another, the function signatures typically have to be the same because the parameters are already on the stack. This severely limits the flexibility to abstractly connect components of the system without having to know the correct semantics beforehand. If the adaptation code specifies the size and type of the parameters of both the original function and the new function the programmer can overcome this barrier by simply (1) calculating the difference in lengths of the two parameter lists, (2) adjusting the frame pointer by the difference in lengths, and (3) copying the new parameters onto the stack.

4 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.

4.1 Patching the Original Code

Our approach is to perform the necessary code modification over the GCC assembler output generated from the source before it is 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.

Since 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 persistency 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.

4.2 Writing Handler Code for an Adaptation Scenario

After the original code is patched to intercept specific function calls, the user of ACC needs to write the handler code to implement any adaptive operations that will be performed at the trapped calls; the user 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, will need to 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, a data structure that keeps track of an intercepted function and stores its accompanying information such as the address of the original arguments.

acc_get_param takes the acc_invoke_entry instance and returns a pointer to pos^{th} input parameter to the original function, allowing the user to access and modify its value. Macros for different data types are also provided for easy access. For example, ACC_GET_PARAM_DOUBLE(ie, pos) returns the value of 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_invoke_func_t is a type used in ACC for generic functions.

While the behavior of a function is changed through intercepting function calls, global variables that are not 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.

4.3 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 needs to be specified so that ACC can determine the correct memory offsets for the parameters.

These initialization steps can be performed in the main function of the glue code if the original code does not 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/last element, respectively, which signifies the local subarray to sort. Then, the code below shows the initialization for dynamically intercepting the sort calls and accessing its parameters using the ACC APIs.

```
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 qsort 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);
```

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 the above code. As described earlier, only primitive data types 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. In the same manner, ACC_HT_POST can be used to operate like the after advice in AOP.

The programmer can implement a desired adaptation logic in the body of the sort_handler function for adapting the original sort function.

5 Case Study 1: Adaptive SOR

We applied ACC to the SOR (Successive Over-Relaxation) routine found in ITPACK 2C [11], 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

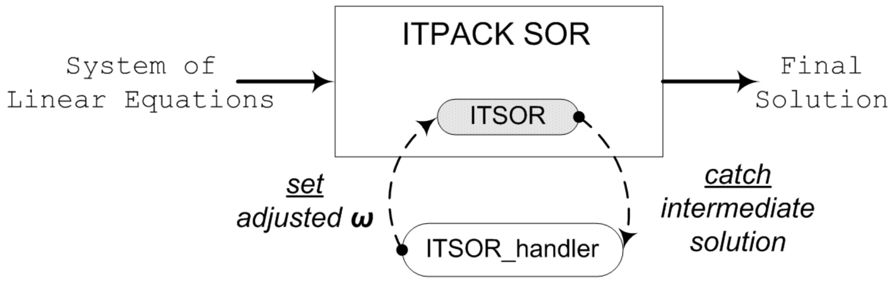


Figure 2: Adaptive SOR system through interception of ITSOR calls using ACC

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 that cannot take advantage of problem-specific information that may 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. In this section 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 is not 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 Hovland and Heath [12] that applies automatic differentiation to a Fortran SOR code to adjust the ω parameter.

5.1 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 (Figure 2). 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, since 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.

In order 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 adaptive SOR code to the two dimensional 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-differences, 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 $1 \le i \le N$, $h = 1/(N+1)$, 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 norm for the Poisson problem at the k^{th} 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 need to be addressed in implementing adaptive SOR: when to change ω and how to obtain its new value. In order to decide the right time to change ω, we simply borrow ITPACK's scheme based on convergence rate estimation [13] and implement it in our handler code. Since 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 scheme suits our needs well. Once it is decided to change ω, we fetch the intermediate solution vector from the parameter list of ITSOR and calculate the residual norm r_k, which is then used in a simple method to chose the new value of ω:

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

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 secant iteration 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 $r'_{k+1}(\omega)$). Since 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.

5.2 Experimental Results

We use $f(x, y) = 0$ and constant Dirichlet boundary conditions for the Poisson problem. We apply the ITPACK SOR solver to the discretized problem with our adaptivity framework turned on and the internal adaptivity capability of ITPACK turned off. The iteration terminates according to ITSOR's stopping test as described in [13].

On a 32 bit, x86 Linux machine running Fedora Core 6 with an Intel Pentium D dual-core 3.60GHz CPU and 2GB 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 known analytically to be

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

Table 2 shows both the execution time (average of 3 runs) and the number of iterations taken to converge for different problem sizes ranging from $N = 300$ to $N = 1000$, 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. We see that our adaptive SOR performed better than ITPACK's in all cases, with speedup as high as 36% in terms of execution time.

In addition, we find that using the ACC framework does not cause 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 0.01 seconds on average, i.e., less than 0.7%.

Figure 3 (top) shows the adaptive progress of ω over iteration 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 only show the last five to highlight the differences between our method and the default ITPACK scheme, and because our first two adaptive steps actually use the ITPACK scheme because two

N	ω_{opt}	ζ	Fixed ω=ω_{opt}		ITPACK		Ours		Speedup
			N_{iter}	time(sec)	N_{iter}	time(sec)	N_{iter}	time(sec)	(time)
300	1.979		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	10^{-2}	493	3.62	1087	7.88	802	5.92	34%
750	1.992		866	14.2	1579	25.7	1268	20.9	23%
1000	1.994		1189	34.4	2222	63.1	1884	53.0	19%
300	1.979		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	10^{-3}	717	5.33	1279	9.30	988	7.24	28%
750	1.992		1172	19.2	1855	30.3	1549	25.4	19%
1000	1.994		1590	46.4	2848	82.1	2281	66.1	24%
300	1.979		555	1.39	859	2.19	728	1.95	12%
400	1.984		746	3.45	1117	5.16	813	4.10	26%
500	1.987	10^{-4}	943	6.94	1495	10.88	1268	9.23	18%
750	1.992		1503	24.6	2193	36.0	1788	30.2	19%
1000	1.994		2003	58.1	3599	103.1	2694	77.0	34%

Table 2: Comparison of adaptive SOR algorithms for Poisson problems

initial points are needed by our estimator. The figure shows that our adaptivity method adjusts ω more quickly toward ω_{opt} than ITPACK, although it overestimates a bit at the end. While the execution time improves by 28%, the handler function for ITSOR consumes only 0.044 seconds on average during the whole execution, which amounts to 0.6% out of the total 7.24 seconds. Figure 3 (bottom) shows the reduction in residual as the computations proceed. (The residuals were computed explicitly at each iteration in a separate run.) We see that our adaptive approach succeeds in improving the rate of residual reduction compared to ITPACK's adaptive scheme. Note that 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.

6 Case Study 2: Parallel CFD Codes

We have used the ACC framework to implement a variety of adaptivity scenarios in the context of real scientific computing codes, including applications in biochemical network simulation [14] and computational fluid dynamics (CFD) [15,16].

In [15] we describe how ACC is used to improve the performance, stability and accuracy of GenIDLEST [17], a large parallel CFD code. GenIDLEST is written in Fortran 90 with MPI, and solves the time-dependent incompressible Navier-Stokes and energy or temperature equations. The stability of the GenIDLEST time integrator depends on the time step used, but it is difficult to identify a single adaptation scheme that will automatically and successfully adjust the timestep for the 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 the simulation can be restarted from the last stable state with a smaller time-step. 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 different strategies to be used and easily experimented with, as appropriate.

In [15] we show how one particular set of CFD computations can be stabilized effectively by a

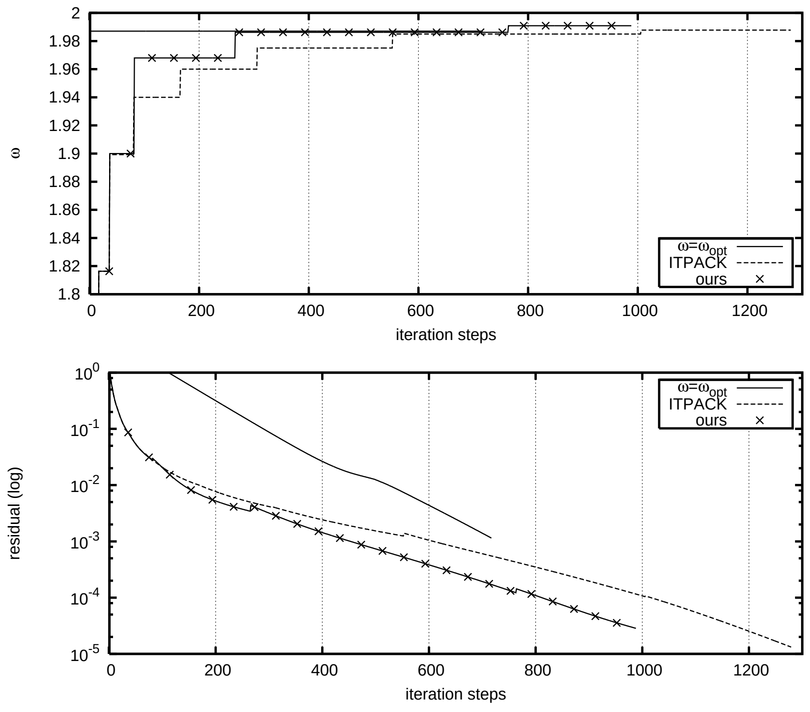
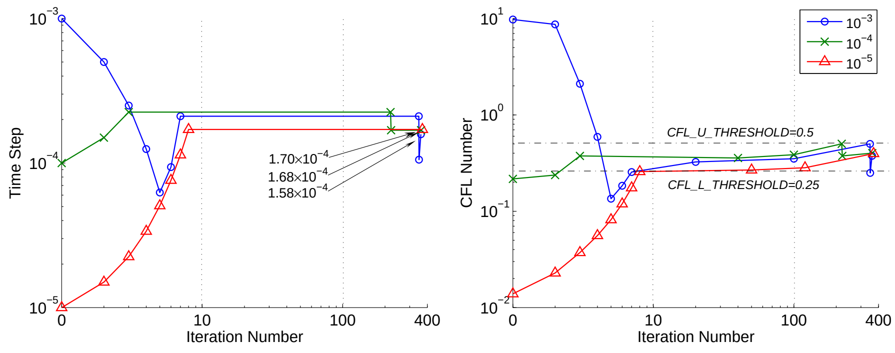


Figure 3: Comparison of SOR methods for a 500×500 Poisson problem



(a) Time Step Change (b) CFL Number

Figure 4: Automatic adjustment of the time step parameter in GenIDLEST

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 CFL (Courant-Friedrich-Levi) indicator goes above or below preset upper and lower CFL thresholds. Figure 4 shows the results for GenIDLEST enhanced with the adaptivity module for a typical simulation, with different initial values of time 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 stability of the simulation.

In addition to automatic time-step adaptation, we also use ACC to facilitate an interactive, user-controlled adaptation of the flow model used in GenIDLEST, 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 is critical to choose an appropriate model when the simulated flow is in the transition region between laminar and turbulent. Our ACC-enabled mechanism allows the user to monitor the variation with time of the stream-wise velocity, 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. Since GenIDLEST already implements all the flow models, with a particular one selected by setting a single parameter, it is easy to use ACC to change models by controlling this parameter, which requires no modification to the original code.

We also are 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 [18], model-driven compiler optimizations [19], and empirical search-based schemes [20]. While these methods have been successful in limited domains (e.g., numerical linear algebra kernels), there is still a need for better support for application-specific adaption schemes, where the unique context of a particular computation—including the code, the data, and the computing resource—can be taken into account.

In [16] we show how ACC can be used to implement a dynamic method for tuning algorithmic parameters in codes such as GenIDLEST. For example, we insert 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 the triple (nb_i, nb_j, nb_k). This parameter defines the structure of the domain decomposition preconditioner; it influences, often in nonobvious ways, both the quality of the preconditioner as an approximation to the original discrete PDE operator and the floating point performance of the computation, e.g., through memory hierarchy effects. The second parameter is ns, the number of inner relaxation sweeps used in the multilevel preconditioner.

Figure 5 shows the performance of GenIDLEST for a typical problem for 3000 time steps, where each point corresponds to the elapsed time measured at 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 $\{(nb_i, nb_j, nb_k), ns\}$ are adjusted automatically to test and improve performance as the simulation proceeds. Overall time-to-solution for the full 10000 time step run in this case is improved by 26% over the performance of the non-adapted code, with typical choices of the parameters.

Modern numerical methods often have several such parameters whose 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

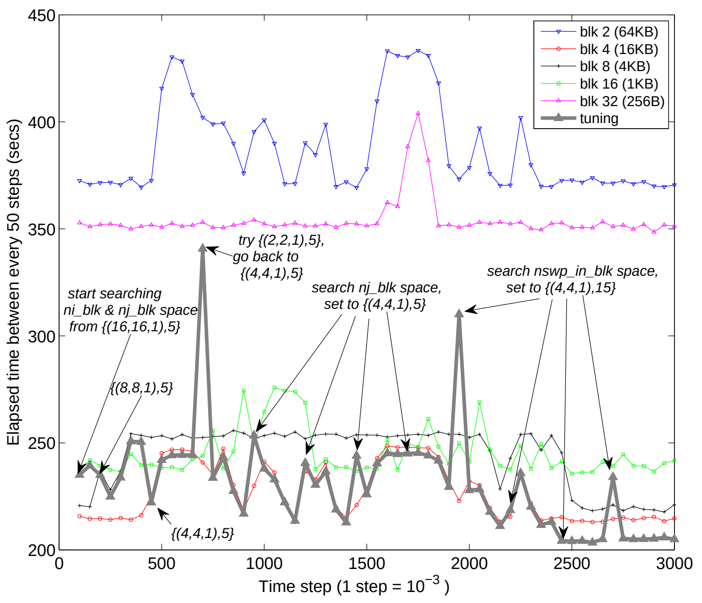


Figure 5: Dynamic tuning of preconditioning parameters in GenIDLEST.

and how to adapt.

7 Conclusion

Our ACC framework is a lightweight function call interception and parameter manipulation library that offers a modular way to extend static scientific code through composition with new program components. Instead of modifying or restructuring the original source code, the compiler-generated assembly output of an original program is patched to transfer the execution control to ACC at the calls to functions a user wants to catch, where the input parameters as well as return values can be accessed. Therefore it allows one to change the program behavior transparently to the original program structure by manipulating the input and output of any function, or by remapping a function to another at the intercepted calls. The overhead from intercepting functions is very small. The ACC framework is therefore well suited to abstracting out new functionality and integrating it with existing non-OO scientific code to realize an adaptive application.

References

[1] 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.* Berlin, Heidelberg, and New York: Springer-Verlag, 1997, vol. 1241, pp. 220–242.

[2] E. Hilsdale and J. Hugunin, “Advice Weaving in AspectJ,” in *AOSD ’04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development.* New York, NY, USA: 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,” in *CRPIT ’02: Proceedings of the 40th Interna-*

tional Conference on Tools Pacific. Darlinghurst, Australia: Australian Computer Society, Inc., 2002, pp. 53–60.

[4] C. Zhang and H.-A. Jacobsen, "TinyC^2: Towards Building a Dynamic Weaving Aspect Language for C," in *Proceedings of the 2nd AOSD Workshop on Foundations of Aspect-Oriented Languages*, Boston, MA, USA, March 2003.

[5] W. R. Mahoney and W. L. Sousan, "Using Common Off-the-Shelf Tools to Implement Dynamic Aspects," *SIGPLAN Not.*, vol. 42, no. 2, pp. 34–41, 2007.

[6] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," in *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation.* New York, NY, USA: ACM Press, 1994, pp. 196–205.

[7] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 4, pp. 317–329, 2000.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation.* New York, NY, USA: ACM Press, 2005, pp. 190–200.

[9] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, 2003.

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

[11] D. R. Kincaid, J. R. Respess, D. M. Young, and R. R. Grimes, "ITPACK 2C: A FORTRAN Package for Solving Large Sparse Linear Systems by Adaptive Accelerated Iterative Methods," *ACM Trans. Math. Softw.*, vol. 8, no. 3, pp. 302–322, 1982.

[12] P. D. Hovland and M. T. Heath, "Adaptive SOR: A Case Study in Automatic Differentiation of Algorithm Parameters," Mathematics and Computer Science Division, Argonne National Laboratory, Tech. Rep. ANL/MCS-P673-0797, 1997.

[13] L. A. Hageman and D. M. Young, Eds., *Applied Iterative Methods.* Academic Press, 1981, ch. 9. The Successive Overrelaxation Method.

[14] P. Kang, Y. Cao, N. Ramakrishnan, C. J. Ribbens, and S. Varadarajan, "Modular Implementation of Adaptive Decisions in Stochastic Simulations," in *SAC '09: Proceedings of the 24th Annual ACM Symposium on Applied Computing.* New York, NY, USA: ACM, March 2009, pp. 995–1001.

[15] 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 *ICCS '09: Proceedings of the 9th International Conference on Computational Science.* Springer, May 2009, pp. 269–279.

[16] ——, "Dynamic Tuning of Algorithmic Parameters of Parallel Scientific Codes," in *ICCS '10: Proceedings of the 10th International Conference on Computational Science*, May 2010, pp. 145–153.

[17] D. Tafti, “GenIDLEST - A Scalable Parallel Computational Tool for Simulating Complex Turbulent Flows,” in *Proceedings of the ASME Fluids Engineering Division (FED)*, vol. 256. ASME-IMECE, November 2001.

[18] C. A. Schaefer, V. Pankratius, and W. F. Tichy, “Atune-IL: An Instrumentation Language for Auto-tuning Parallel Applications,” in *Euro-Par ’09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing.* Berlin, Heidelberg: Springer-Verlag, 2009, pp. 9–20.

[19] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[20] 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*, vol. 93, no. 2, pp. 293–312, Feb. 2005.