Modular Implementation of Adaptive Decisions in Stochastic Simulations

Pilsung Kang, Yang Cao, Naren Ramakrishnan, Calvin J. Ribbens, and Srinidhi Varadarajan

Department of Computer Science
Virginia Tech, VA 24061, USA

{kangp,ycao,naren,ribbens,srinidhi}@cs.vt.edu

ABSTRACT

We present a modular approach to implement adaptive decisions with *existing* scientific codes. Using a sophisticated system software tool based on the function call interception technique, an external code module is transparently combined with the given program *without* altering the original code structure, resulting in a newly composed application with extended behavior. This is useful for generalizing codes into using different parameter values or to switch algorithms or implementations at runtime. Applying the proposed method on a biochemical stochastic simulation software package to implement a set of exemplary use cases, which includes changing program parameters, substituting random number generators, and dynamically changing the stochastic simulation method, we demonstrate how effectively new code modules can be plugged in to construct an application with enhanced capabilities.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming—*program modification, program transformation*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*structured programming*

General Terms

Languages, Management

Keywords

modular composition, program modification, function call interception, stochastic simulation

1. INTRODUCTION

Coping with change is challenging in scientific programming, where old code bases are common and modern programming practices that encourage modularity and adaptability have not always been used (historically). For instance, numerical software more than a few decades old is not unusual in the listings in the Netlib repository [3]. Even though those codes established stability through numerous bug fixes and performance improvements over their lifetime, it has become inflexible to change their code structure too, unless they had been designed for future restructuring from their inception [11]. The fact that a large portion of scientific codes were written in early versions of Fortran adds to their inflexibility with respect to program changes.

In fact, software evolution has long been recognized as an inevitable phenomenon [20, 21]. From the initial design and development process, to management jobs such as fixing bugs and testing functions, to whole-scale updates like restructuring of the entire code base, the programmer needs to cope with changes that are necessary across software generations.

Even for short-term modification tasks such as adding or changing functional behavior, however, implementing adaptive decisions on top of existing code can be an onerous task. In simple cases such as replacing a function call with a conditional control structure, the modification replaces the original call by an `if-then-else` statement where either of two functions is chosen depending on the runtime value of a predicate. Yet the rewriting process may become cumbersome in large programs with a complex adaptive plan because the programmer has to locate and update all the places where the modification is needed. Sometimes it requires restructuring of the whole program, which can be quite imposing. Therefore, this issue of adding new functionality to existing code in a modular way has been recently identified by the object-oriented programming community as one of the important motivations for Aspect-Oriented Programming (AOP) [18].

However, although object-oriented techniques for modular program development are getting more support in the scientific programming community [10], advanced code insertion features like AOP weaving [16] are still lacking.

In this paper, we present a modular method based on function call interception manipulation techniques for implementing adaptive decisions on existing scientific programs, in which the new code is written and managed as a separate module with regard to the original program, thus achieving transparency through the modification process. We chose biochemical stochastic simulation as an application because, since first pioneered by Gillespie [12, 13], new and improved approaches continue to be developed [14, 28, 8], which motivates a diverse set of adaptive scenarios.

The rest of the paper is organized as follows. Section 2 addresses the programming environment for the proposed method and the target simulation software. Section 3 describes a set of adaptive decision scenarios and their individual implementations in detail, with experimental results. Section 4 summarizes related research work. Finally, Section 5 concludes the paper.

2. SETUP FOR IMPLEMENTING ADAPTIVE DECISIONS

To realize a modular way of extending and modifying applications without altering the original code structure, we employ *Invoke*, a runtime function call interception tool. Function call interception (FCI), sometimes referred to as method call interception, is a technique whereby function calls are intercepted in order to alter the operations performed when the call is actually made. With FCI, the desired operations may be performed either just before or after the call, or at both times. Because specific function calls are caught and manipulated at runtime to change their original behavior, FCI enables transparent code modification without directly rewriting the existing code.

2.1 Invoke

Invoke is a compositional framework with a set of APIs which support FCI for x86 architectures [15]. It implements FCI at the assembly language level by replacing the x86 `call` instruction in the instrumentation target code with a call to its own interception handler, establishing call site modifications in the target.

The application development process using Invoke involves three steps. First, the assembly language source code of the target subprogram is patched to divert the target function calls to Invoke. Next, the intended adaptive plan is implemented in a new code module, in which the desired operations are described and the original calls can be modified through Invoke's parameter manipulation APIs. Invoke transfers execution control of the diverted calls to the new module at execution time. Finally, the user needs to set up an association between the target function and the new module by using Invoke's registration APIs.

Since Invoke patches the assembly output generated from the compiler to insert new code, the original high-level language code is not affected, thus enabling modular development of new code modules and transparent modification over existing programs. Moreover, due to the assembly level code modification, Invoke can be applied independently of the original source language, making it suitable for Fortran legacy codes as well as programs written in C or C++.

Figure 1 shows how Invoke operates to combine a new code module with a given program, in which every call to the target function is intercepted and program control is diverted to the associated *handler*, a piece of newly inserted code responsible for modifying the original function, through Invoke's bookkeeping data structure.

Although we say Invoke implements FCI functions, its capabilities are actually quite general including:

- Function interception : User-specified function calls can be intercepted before or after their lifetime through assembly code patching. This means that the basic unit to support code insertion is a procedure or function.

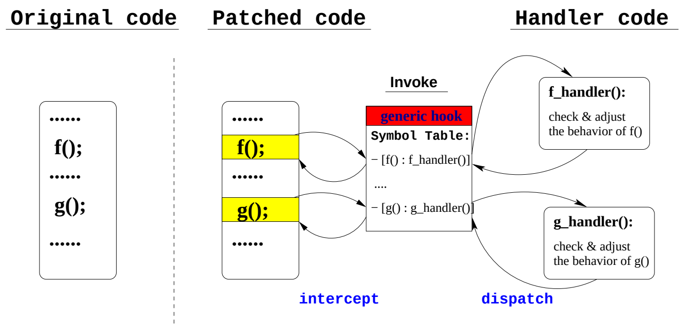

Figure 1: Function call interception using Invoke.

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

The most useful feature of Invoke is the ability to remap the entire parameter list of a function to a new function even with a different signature. We make heavy use of this functionality to implement the adaptive use cases presented in this paper.

2.2 StochKit

StochKit is a stochastic simulation framework written in C++ for simulating biochemical reaction systems [5]. It includes a core set of algorithm implementations from Gillespie's exact stochastic simulation algorithm (SSA), from explicit [14] and implicit τ-leaping [28] methods to trapezoidal τ-leaping [9] methods, offering its users flexible options to control the simulation step function and step size. Although in its beta stage, StochKit also provides MPI interfaces for large scale Monte Carlo simulations on a parallel cluster. StochKit also provides essential data structures such as `Vector` and `Matrix` classes, and integrates external packages like RANLIB [4, 19] and SPRNG 2.0 [24] for generating random numbers from distinct statistical distributions.

Because it is readily accessible and under active development, we use StochKit as a target program to apply Invoke's function call manipulation functionality, effectively extending its behavior in subprograms. Specifically, the modules for random number generation and the exact SSA simulation are prime candidates for program modification purposes and are covered in the next section.

3. ADAPTIVE SCENARIOS AND THEIR IMPLEMENTATIONS

In this section, we present three adaptive scenarios in the context of using StochKit and their corresponding implementations under the Invoke framework. Through catching and manipulating specific function calls, the examples show how an adaptive decision can be applied transparently to an existing code base **without** altering the original source code.

The first two examples illustrate a static adaptive scenario where a subprogram module is replaced with an alternative package, which can be considered as an effective interim solution to updating the StochKit package without an explicit version update. The last example presents a dynamic case in which the simulation switches the stochastic simulation models based on 'hints' supplied by the user.

3.1 Change of Program Parameter Values

One way of changing the program behavior is to modify program states stored in variables. Using Invoke, global variables can be accessed by external declarations in a new code, in which their values are adjusted at appropriate times, once the target functions to intercept are determined and proper control points are defined. In addition, input parameters to those functions are accessible as well and their values can be altered at will through Invoke's stack manipulation APIs. The local variables not communicated as function arguments and certain program parameters hardwired as constants are not modified by external code modules. However, the effect of changing local values can be obtained by intercepting the function block that encloses those local variables, and calling instead a substitute function which uses different values for those variables. The following code shows an example of such a case in which the values of preset threshold parameters are altered.

Listing 1: StochKit `PoissonRandom` Function

```
 1  double PoissonRandom(double mean)
    {
 3    if (mean == 0) {
        return 0;
 5    } else if (mean > 1e6) {
        return mean;
 7    } else if (mean > 1e4) {
        return (mean + sqrt(mean)*snorm());
 9    } else
        return ignpoi(mean);
11  }
```

Listing 1 shows the original StochKit `PoissonRandom` function, which is used for the τ-leaping SSA method [14] to generate random numbers from the Poisson distribution. It uses hardwired values, 10^6 and 10^4 (line 5 and 7), to branch into different approximations of the Poisson distribution depending on how large the input `mean` is.

Listing 2: Change of Hardwired Parameter Values

```
 1  void PRandom_handler(struct li_invoke_entry *ie,
                         LI_HANDLER_TYPE type)
 3  {
      if (type == LI_HT_PRE) {
 5      /* remap to our version of PoissonRandom*/
        li_remap(ie, PoissonRandomNew);
 7    }
    }

    double PoissonRandomNew(double mean)
11  {
      if (mean == 0) {
13      return 0;
      } else if (mean > 1e7) {//originally 1e6
15      return mean;
      } else if (mean > 1e5) {//originally 1e4
17      return (mean + sqrt(mean)*snorm());
      } else
19      return ignpoi(mean);
    }
```

Listing 2 is an implementation of changing the threshold values in `PoissonRandom`. In order to implement an adaptive plan of using different values for the local parameters, `PRandom_handler` catches every call to the target `PoissonRandom` function at its beginning (line 4) and by calling `li_remap` (line 6), Invoke's function call remapping API, `PRandom_handler` entirely replaces it with `PoissonRandomNew`, in which new parameter values (10^7 and 10^5), replacing old values, are used for conditional branching out.

3.2 Substitution of Random Number Generator Libraries

Any moderate-sized stochastic simulation needs a good random number generator (RNG)—good in terms of both the speed and the quality of randomness of the generated numbers. For instance, while the C `random` function returns a number in the range from 0 to $(2^{31} - 1)$ with the approximate period $16 \times (2^{32} - 1)$, the Mersenne Twister MT 19937 [25] generates a number in a doubled range with the extremely large period $(2^{19937} - 1)$. Thus, programmers of stochastic simulations often want to upgrade and use more up-to-date or optimized versions of the RNG package of their choice, or switch to a new, more efficient, package. However, updating a program package to include a new library may be cumbersome since combining multiple packages can involve major rewriting of the original program, including new headers, redefining macros, and adding and resolving dependencies in project build files. In fact, what is desired in this case may be just changing one function, i.e., the RNG function. Thus, a function call interception technique can offer an easy way of plugging in new code into an existing software code base, as the following example demonstrates.

Listing 3: Substitution of RNGs

```
 1  /* handler for standard C random() calls */
 2  void CRandom_handler(struct li_invoke_entry *ie,
                         LI_HANDLER_TYPE type) {
 4    if (type == LI_HT_PRE) {
    #ifdef USE_SPRNG /* redirect to ISprngRandom */
 6      li_remap(ie, ISprngRandom);

 8  #elif defined(USE_SFMT) /* redirect to SFMT */
        li_remap(ie, ISFMTRandom);

    #endif
12    }
    }

    #ifdef USE_SPRNG
16  long int ISprngRandom() {
      return (long) isprng();
18  }

20  #elif defined(USE_SFMT)
    long int ISFMTRandom() {
22    return (long) (gen_rand32()>>1);
    }
24  #endif
```

Listing 3 shows a code snippet for a handler function associated with the standard C `random` calls performed in StochKit. Assembly patching needs to be applied to only one file, `Random.cpp`, since the C `random` is used only in the StochKit interfaces defined in `Random.cpp`. The user may have just one RNG package of his choice and always opt for it by catching and remapping C `random` calls to it. The code listing shows other cases where multiple RNGs are available such that, depending on the user's selection (line 5,8) at compile time, either Mersenne Twister SFMT [29] or SPRNG 2.0 [24] bundled in the StochKit distribution is substituted for `random` (line 16,21).

3.3 Switching Stochastic Simulation Models

The slow-scale SSA (ss-SSA) [7, 8] is an efficient approximation method of the exact SSA which exploits the fact that a certain kind of "fast" reactions are much less significant to the system's evolution than the "slow" reactions, thereby stochastically simulating only slow reaction events to advance the system in time. To apply the ss-SSA, however, the programmer needs to have *a priori* knowledge about the simulated system regarding what reaction channels are slow and fast, something that can be determined only after executing simulations. We present a way to overcome this constraint through dynamic simulation model change using Invoke's function call catch and substitution capability. The adaptive procedure takes the following steps.

1. Start the simulation with the exact SSA.
2. The execution pauses after a preset time, showing the intermediate simulation result for analysis of the system, and prompts for the user's intervention.
3. With the understanding of the system, the user identifies fast reactions and inputs them.
4. The simulation continues the exact SSA, but effectively considering only slow reactions due to the modified propensity function that uses the ss-SSA approach based on the user's hint.

Ideally, an online recommender system can be conceived that monitors the simulation progress and automatically makes the algorithm switching decisions based on the runtime analysis. However, the design of recommender systems is a major subject by itself [17, 27], and outside the scope of this paper. Instead, we assume a human with domain knowledge in the simulation loop for making adaptive decisions to trigger the switching in simulation methods.

The goal is to realize the ss-SSA algorithm upon the existing StochKit SSA code through the Invoke framework to manipulate the original function calls to set up the ss-SSA style execution control. Following the above scenario, the adaptive control points map to two functions in StochKit. `SSA_SingleStep` needs to be intercepted to stop the program execution after a certain time in the beginning of the simulation and prompt the user with interim results. In addition, the `Propensity` function calls need to be handled to effectively zero out the probabilities of fast reaction channels to model the ss-SSA method in continuing the simulation. Since the two functions are called by the same `StochRxn` function in `StochRxn.cpp`, the patching is only applied to the file, thereby opening room for inserting new code that contains a handler for each of the target functions.

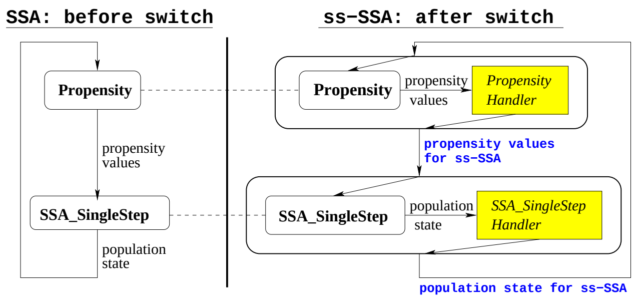

Figure 2: Exact SSA extended to ss-SSA through Invoke.

Figure 2 shows how the original SSA functions in StochKit are arranged with the accompanying handlers to attain the intended ss-SSA behavior. `SSA_SingleStep_Handler` performs two tasks depending on whether it operates before or after `SSA_SingleStep` executes. As a pre-handler, it accesses `current_time` placed as the second argument of the `SSA_SingleStep` call through Invoke's parameter accessing API and checks if the program state has reached `PAUSE_TIME`, when it will stop to show the simulated system states and wait for the domain expert to analyze the data and set which reaction channels are fast. As a post-handler, it does nothing until the system reaches `PAUSE_TIME`, after which it cooperates with `Propensity_Handler` to calculate the population of the system to mimic ss-SSA while maintaining the original execution flow of the exact SSA implementation in StochKit. Specifically, `SSA_SingleStep_Handler` computes an analytical solution for the population of each fast variable that has been previously identified by the user.

`Propensity_Handler` operates only after `PAUSE_TIME`. It simply remaps the original `Propensity` function call to new code, `PropensityNew`, in which only the slow variables are considered for stochastic simulation based on the theory developed in [7, 8]. The following examples show experimental results of the ss-SSA implementation under Invoke for a certain set of fast reaction types.

3.3.1 *The Fast Reversible Isomerization*

Following an example described in [8], we consider the fast reversible isomerization,

$$S_1 \underset{c_2}{\overset{c_1}{\rightleftarrows}} S_2,$$

occurring together with a single slow reaction,

$$S_2 \overset{c_3}{\rightarrow} S_3,$$

where S_i denotes the species in the system and c_j is the rate constant for the reaction channel R_j. Given the system's state vector $X(t) \equiv (X_1(t), X_2(t), X_3(t)) = \mathbf{x}$, where $X_i(t)$ is the number of molecules of S_i at time t, the following propensity functions $a_j(\mathbf{x})$ and state-changing vectors ν_j define the system's stochasticity,

$$\begin{aligned} a_1(x) &= c_1x_1, & \nu_1 &= (-1, +1, 0), \\ a_2(x) &= c_2x_2, & \nu_2 &= (+1, -1, 0), \\ a_3(x) &= c_3x_2, & \nu_3 &= (0, -1, +1). \end{aligned}$$

The ss-SSA substitutes the approximate analytical solution obtained by solving the stationary deterministic reaction

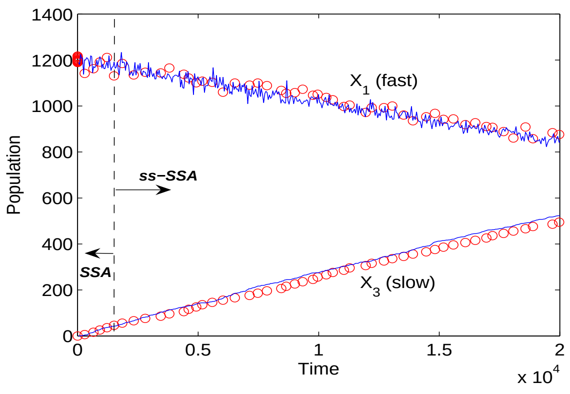

Figure 3: Time evolution of ss-SSA under Invoke (red circle) vs. exact SSA (blue solid) for the fast reversible isomerization. The dashed line indicating the time of model change is shown farther from its actual location (PAUSE_TIME=1.0) for ease of illustration.

rate equation for fast species at each exact SSA step,

$$x_T = x_1 + x_2, \quad x_1 = \frac{c_2 x_T}{c_1 + c_2}, \quad x_2 = x_T - x_1.$$

The ss-SSA implemented with Invoke, along with the exact SSA for comparison, was carried out with the following set of parameter values,

$$c_1 = 1, \quad c_2 = 2, \quad c_3 = 5 \times 10^{-5},$$
$$x_1 = 1200, \quad x_2 = 600, \quad x_3 = 0 \quad \text{at } t = 0.$$

Figure 3 shows how the implemented ss-SSA closely matches SSA by comparing the time evolution of both the fast and slow variables of the simulated system obtained from a single run of each method. The interactive ss-SSA implementation was programmed to stop to accept the user's input quite early at $t = 1.0$, when the simulation has taken only a couple of thousands SSA steps out of a dozen million that the exact SSA would take.

Figure 4 and Table 1 show the statistical properties of the ss-SSA compared with the SSA through 10,000 samples. The ss-SSA code was run on a Linux machine with an AMD Athlon X2 4000+ dual core processor and 2GB memory, whereas the SSA was on a 50 node Linux cluster, each with an AMD Opteron 240 dual core processor and 1GB memory, resulting in 100 MPI processes responsible for 100 samples per process. The probability density plots of the ss-SSA appear to be statistically indistinguishable from the exact SSA's in both the fast and slow reactions, which is backed in the comparison table in which the mean and the standard deviation of each species simulated by the ss-SSA are virtually the same as those obtained from the SSA. In contrast, the gain in computational efficiency is huge as expected. The ss-SSA took only 16 seconds to complete 10,000 samples, while the exact SSA took about an hour for each MPI process on the cluster.

3.3.2 The Enzyme-Substrate Reaction

As another example for the ss-SSA, the simple enzyme-substrate reaction process has been tested. The reaction in which the enzyme S_1 combines with the substrate S_2 to produce an intermediate unstable enzyme-substrate complex S_3

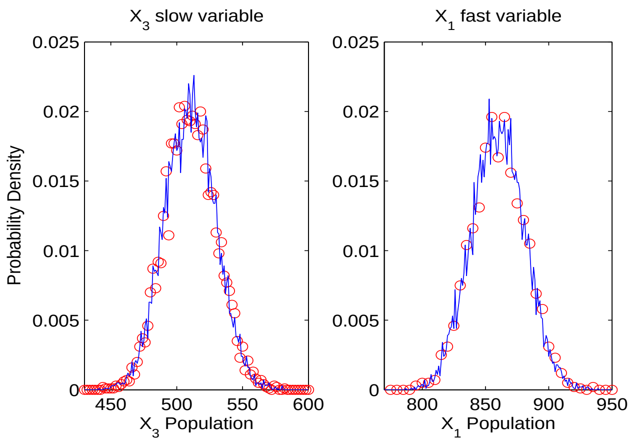

Figure 4: Probability densities of ss-SSA under Invoke (red circle) vs. exact SSA (blue solid) for the fast reversible isomerization.

	X_1 (fast)		X_2 (fast)		X_3 (slow)	
	Mean	Std	Mean	Std	Mean	Std
SSA	859.86	20.99	429.97	18.00	510.18	19.35
ss-SSA	859.80	21.35	429.88	17.90	510.32	19.26

Table 1: Statistics of ss-SSA under Invoke vs. exact SSA for the fast reversible isomerization.

is modeled as a reversible fast process,

$$S_1 + S_2 \underset{c_2}{\overset{c_1}{\rightleftarrows}} S_3,$$

while S_3 decays into its converted constituents S_1 and the product S_4 very slowly ($c_2 \gg c_3$),

$$S_3 \overset{c_3}{\rightarrow} S_1 + S_4.$$

Thus, the system is represented by the following propensity functions and state-changing vectors,

$$\begin{aligned} a_1(x) &= c_1 x_1 x_2, & \nu_1 &= (-1, -1, +1, 0), \\ a_2(x) &= c_2 x_3, & \nu_2 &= (+1, +1, -1, 0), \\ a_3(x) &= c_3 x_3, & \nu_3 &= (+1, 0, -1, +1). \end{aligned}$$

The derivation of approximate analytical solutions for the fast variables, x_1, x_2, and x_3, to substitute for the exact SSA simulation is described and discussed in detail in [7], to which we refer the reader, and show only the experimental application results in this paper due to space limitations.

The ss-SSA was implemented through Invoke's function call interception and remapping capability in the same manner as in the fast reversible isomerization example, in which the calls to the `SSA_SingleStep` and `Propensity` functions are manipulated to make them behave according to the ss-SSA method.

Figure 5 plots the time evolution of one fast variable (x_2) and the slow variable (x_4) in one realization instance, showing the close approximation achieved by the ss-SSA implementation over the exact SSA. Table 2 shows the statistical data obtained from 10,000 samples by running the adaptive ss-SSA program, as well as the exact SSA data for comparison, using the following set of parameter values where

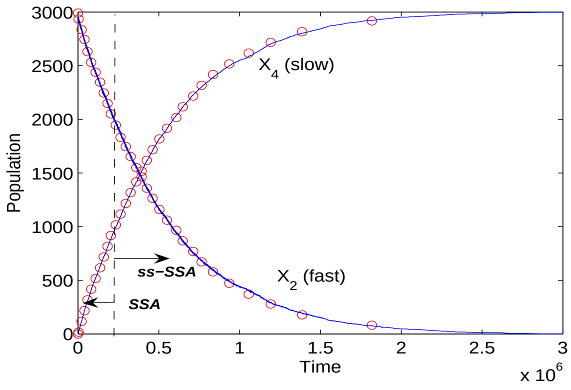

Figure 5: Time evolution of ss-SSA under Invoke (red circle) vs. exact SSA (blue solid) for the enzyme-substrate reaction. The dashed line indicating the time of model change (PAUSE_TIME=1.0) is shown farther from its actual location for ease of illustration.

	X_1 (fast)		X_2 (fast)		X_3 (fast)		X_4 (slow)	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
SSA	220.00	0.04	0.08	0.29	0.00	0.04	2999.91	0.29
ss-SSA	220.00	0.03	0.08	0.29	0.00	0.03	2999.92	0.29

Table 2: Statistics of ss-SSA under Invoke vs. exact SSA for the enzyme-substrate reaction.

$c_3 = 5 \times 10^{-5}$ is set much smaller than $c_2 = 2$.

$$c_1 = 1, \quad c_2 = 2, \quad c_3 = 5 \times 10^{-5},$$
$$x_1 = 1200, \quad x_2 = 600, \quad x_3 = 0 \quad \text{at } t = 0.$$

As in the reversible isomerization case, the adaptive method that switches from SSA to ss-SSA generated statistically indistinguishable data in the enzyme-substrate reaction simulation.

In summary, the adaptive approach through Invoke achieves extreme performance boost, as shown in Table 3 that compares the execution time of the implemented ss-SSA on a single machine to that of the exact SSA using 100 processors on a cluster.

4. RELATED WORK

Function call interception techniques are typically used for debugging purposes and performance analysis such as tracing and profiling the dynamic execution flow of a program. Additionally, they are also used to change program behavior by modifying input parameters or the return value of the intercepted function. Some techniques even enable users to replace entirely a function with an alternative implementation. The GNU C/C++ compiler [2] can generate instrumentation calls for entry and exit to functions through the `-finstrument-functions` compiler option. However, owing to a lack of sophisticated parameter manipulation and function redirection capabilities as in Invoke, this compiler option only serves for profiling purposes and cannot be used to change the original behavior of the instrumented functions.

In Aspect-Oriented Programming (AOP) frameworks, FCI is used to implement *advice weaving* [16] on a set of preset control points, which are defined to be an entry or exit of functions with the same concern, such that a piece of associated code is inserted to execute whenever the points are reached by the program in execution. 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 [1, 30, 32, 23]. However, comparable support for Fortran, which has been widely used for implementing scientific codes, is still unsupported. In contrast, the proposed method offers a language-neutral way of overlaying new code onto existing programs due to its assembly level code patching.

Binary instrumentation tools can overcome this language dependency issue. These tools insert code into 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 and the accompanying overhead is typically too significant, most of them are developed for rigorous programs analysis purposes [31, 6, 22, 26] such as debugging and profiling rather than as a tool to aid programmers in extending existing programs in a modular way. Invoke, unlike fine-grained binary instrumentation tools, is coarse-grained, having function blocks as the operation unit, thus enabling lightweight code insertion.

	Fast Reversible Isomerization	Enzyme-Substrate Reaction
SSA (100 procs cluster)	3665 sec	5810 sec
ss-SSA (single machine)	16 sec	46 sec

Table 3: Execution Time Comparison of ss-SSA under Invoke vs. exact SSA.

5. CONCLUSIONS

Sophisticated function call interception and manipulation tools such as provided by Invoke offer a modular way to extend the behavior of existing programs without explicitly rewriting the original source code. As demonstrated through a set of examples including the change of program parameters, substitution of random number generators, and dynamic switching of stochastic simulation methods, transparent insertion of external modules enables composition of new applications from the existing code, thereby implementing adaptive decisions that necessarily occur during software lifetimes. In particular, considering scientific codes are traditionally inflexible to update, the function call interception technique can be effectively employed to deliver the modular realization of complex adaptive decisions.

6. REFERENCES

[1] ACC. `http://www.aspectc.net`.

[2] GNU Compiler Collection. `http://gcc.gnu.org/onlinedocs/`.

[3] Netlib. `http://www.netlib.org`.

[4] RANLIB C Implementation. `http://www.netlib.org/random`.

[5] StochKit. `http://www.engineering.ucsb.edu/~cse/StochKit`.

[6] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[7] Y. Cao, D. T. Gillespie, and L. R. Petzold. Accelerated Stochastic Simulation of the Stiff Enzyme-Substrate Reaction. *Journal of Chemical Physics*, 123(14):144917.1–144917.12, October 2005.

[8] Y. Cao, D. T. Gillespie, and L. R. Petzold. The Slow-Scale Stochastic Simulation Algorithm. *Journal of Chemical Physics*, 122(1):014116, January 2005.

[9] Y. Cao and L. R. Petzold. Trapezoidal Tau-Leaping Formula for the Stochastic Simulation of Biochemical Systems. *Proceedings of Foundations of Systems Biology in Engineering (FOSBE)*, pages 149–152, 2005.

[10] V. K. Decyk, C. D. Norton, and H. J. Gardner. Why Fortran? *Computing in Science and Engineering*, 9(4):68–71, 2007.

[11] P. F. Dubois. Ten Good Practices in Scientific Programming. *Computing in Science and Engg.*, 1(1):7–11, 1999.

[12] D. T. Gillespie. A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions. *Journal of Computational Physics*, 22(4):403–434, December 1976.

[13] D. T. Gillespie. Exact Stochastic Simulation of Coupled Chemical Reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[14] D. T. Gillespie. Approximate Accelerated Stochastic Simulation of Chemically Reacting Systems. *The Journal of Chemical Physics*, 115(4):1716–1733, 2001.

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

[16] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[17] E. N. Houstis, A. C. Catlin, J. R. Rice, V. S. Verykios, N. Ramakrishnan, and C. E. Houstis. PYTHIA-II: A Knowledge/Database System for Managing Performance Data and Recommending Scientific Software. *ACM Trans. Math. Softw.*, 26(2):227–253, 2000.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[19] P. L'Ecuyer and S. Côté. Implementing a Random Number Package with Splitting Facilities. *ACM Trans. Math. Softw.*, 17(1):98–111, 1991.

[20] M. M. Lehman. The Programming Process. *Technical Report RC2722, IBM Research Report*, December 1969.

[21] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[22] 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*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[23] W. R. Mahoney and W. L. Sousan. Using Common Off-the-Shelf Tools to Implement Dynamic Aspects. *SIGPLAN Not.*, 42(2):34–41, 2007.

[24] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation. *ACM Trans. Math. Softw.*, 26(3):436–461, 2000.

[25] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

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

[27] N. Ramakrishnan, E. N. Houstis, and J. R. Rice. Recommender Systems for Problem Solving Environments. In H. Kautz, editor, *Working Notes of the AAAI-98 Workshop on Recommender Systems*, pages 91–95. AAAI/MIT Press, 1998.

[28] M. Rathinam, L. R. Petzold, Y. Cao, and D. T. Gillespie. Stiffness in Stochastic Chemically Reacting Systems: The Implicit Tau-Leaping Method. *The Journal of Chemical Physics*, 119(24):12784–12794, 2003.

[29] M. Saito and M. Matsumoto. SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer Berlin Heidelberg, 2006.

[30] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *CRPIT '02: Proceedings of the 40th International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[31] 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*, pages 196–205, New York, NY, USA, 1994. ACM Press.

[32] 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.