A Runtime Framework for Adaptive Compositional Modeling

Michael Alan Heffner
Department of Computer Science
Virginia Tech, Blacksburg, VA 24061

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Computer Science and Applications

Examining Committee:

Srinidhi Varadarajan, Co-Chair
Naren Ramakrishnan, Co-Chair
Calvin J. Ribbens

May 7, 2004
Blacksburg, Virginia

Keywords: Adaptive Compositional Modeling, Runtime Framework, Object-Level Patching

A Runtime Framework for Adaptive Compositional Modeling

Michael Alan Heffner

Abstract

The rapid emergence of embedded devices and sensor networks that frequently exchange object-level images foretells an increasing reliance on object-level systems. Additionally, nearly all computing systems, including control systems, enterprise applications, scientific codes and dynamic libraries operate eventually at the object code level. Studying adaptivity and runtime composition issues in such systems is becoming an important focus of systems research. In this thesis, we describe an object-level framework that will manipulate an object module to instrument control functionality and adaptivity in order to realize complex compositional scenarios. Using function and parameter remapping capabilities, our framework transcends programming language and design boundaries, and enables applications to adapt dynamically during runtime. We introduce the capability to "restart" an application automatically, a feature we utilize to support adaptivity not only spatially, over the algorithm domain, but temporally as well. A high-level adaptive control language based on XML is presented that allows complex adaptive scenarios to be expressed concisely. Additionally, the construction of several adaptive scenarios using our framework is illustrated, along with several experiments in "learning adaptivity" using reinforcement learning techniques.

Acknowledgements

First, I would like thank my advisors Dr. Naren Ramakrishnan and Dr. Srinidhi Varadarajan. I would like to thank Dr. Ramakrishnan for his immense help on proofreading the thesis document, providing me with the examples for the reinforcement learning scenarios, and convincing me that I could graduate on time. Dr. Varadarajan introduced me to this topic and, by letting me experiment with the Weaves framework, the concept of object-level manipulation.

Second, I would like to thank all the graduate students (Bharath, Joe, Muthukumar, Ashwin, Dan, and others) in the 2160 lab that provided numerous opportunities for political, philosophical and technical discussions. I would also like to thank my roommates and my friends for the occasional opportunity to escape from work.

Last, but not least, I would like to thank my parents Alan and Jen and the rest of my family for their continued support and encouragement throughout my academic career.

Contents

1 Introduction **1**

2 Basic Constructs for Adaptive Compositional Modeling **4**

2.1 Primitives for Adaptive Compositional Modeling 4
2.1.1 Function interception 4
2.1.2 Registered callbacks 5
2.1.3 Parameter manipulation 5
2.1.4 Dynamic process rollback 6
2.2 Usage Scenarios 7
2.2.1 Scenario 1: Switching from Quicksort to Insertionsort 7
2.2.2 Scenario 2: DQAG Parameter Sweeping 8
2.2.3 Scenario 3: Putting it all together 9
2.3 High-level Adaptive Control Language 11
2.3.1 Parameter Mappings Section 11
2.3.2 Callback Section 12

3 Framework Performance Evaluation **15**

3.1 Benchmarking 16
3.2 Learning to be Adaptive 17
3.2.1 Sorting using Two Algorithms 18
3.2.2 Designing an offline recommender for DQAG 19
3.2.3 Designing an online recommender for DQAG 21

4 Related Work **26**

4.1 Post-Object Programming Mechanisms 26
4.1.1 Aspect-Oriented Programming 26
4.1.2 Generic Programming 27
4.2 Tunability Interfaces 27
4.3 Compositional Modeling 28
4.4 Binary Interception 29

5 Discussion **30**

5.1 Future Work 31

A HACL DTD **32**

List of Figures

3.1 HPL Performance with Framework. 16
3.2 Sweep3D Performance with Framework. 17
3.3 Comparison of total operations between quicksort and insertionsort. 18
3.4 Color plot showing distribution of optimal *KEY* values for DQAG usage. 20
3.5 Example illustration of the operation of DQAG(E). 21
3.6 Comparison of fixed point quadrature rules and random policy. 22
3.7 Plot of required evaluations for differing probabilities of quadrature rule increase. . 23
3.8 Comparison of all the RL policies we tested. 24

5.1 Illustration of patching procedure. 31

List of Tables

2.1 Example of parameter mappings from quicksort scenario. 12
2.2 Example of callback section in HACL XML document. 13

3.1 Comparison of execution times for an empty function. 15

Chapter 1

Introduction

High-performance scientific codes permeate many engineering and business contexts. While such codes today are primarily meant to run on desktops and clusters, it has been predicted that over 98% of computing power will be harnessed in embedded devices and sensor networks by the year 2010. Across this broad range of diversity — control systems, enterprise applications, embedded operating environments, and dynamic libraries — and additionally, across the list of different programming languages and conceptual software models, there is a single common factor: the object code that is being executed on the system. This common factor provides an opportunity for enormous exploitation of functionality across the multitudes of codes and implementation techniques used to construct them. A framework that will manipulate an object module at runtime to instrument control functionality and adaptivity will hence prove very useful across many domains and will be relevant for an extended period of time.

For instance, a framework that instruments control on object files after execution has started can be used to develop applications that exploit minimal bootstrapping but which are fully extensible at runtime. In such an application, only the most essential components are loaded at runtime — the object-level framework intercepts calls to components not yet loaded and dynamically links them into the application just in time for their execution. These components are essentially brought in using a very-late binding approach that permits off-line code optimization and just-in-time compilation [3]. Similarly, the opposite can occur in which object modules are disconnected during runtime or they can be replaced at runtime with newer versions, a feature that permits long-running applications to adapt and mutate over time. An object-level framework can transparently massage type differences across components, which opens the door to a range of compositional modeling [1] applications [2, 10] that can now compose systems of seemingly arbitrary collections of objects and functionality. These models bridge the implementation language barriers [19] that typically limit the functionality that is available to a single code and they can also facilitate concern [9] and concept [24] based programming paradigms. The cross-object transparent invocation functionality can also be exploited in distributed dynamic systems [36]; for example, it can be used to implement many of the components of the CORBA system [13] and Microsoft's Component Object Model (COM) [30]. Object-level functionality is also applicable to sensor networks that require embedded mobile agents to dynamically reconfigure themselves many times during runtime, typically requiring object code to be transfered across multiple heterogenous environments.

There is also a range of security applications that can benefit from an object-level framework. Installations that require detailed accounting and auditing statistics can track these application

parameters at the object code level, recording information such as function call sequences, system call parameters, and time spent performing specific routines. Additionally, such a framework could facilitate dynamic security patching at runtime by diverting calls away from previously vulnerable methods to their new fixed counterparts, without requiring large servers to be taken down for costly downtime-patching. Such a framework could also be used to harbor potentially naive bodies of code by intercepting intentionally malformed parameter values. If an object-level framework also supports adaptivity, it enables a range of applications such as micro electro-mechanical systems (MEMS), resource tunability [7, 12, 17, 23, 29, 37], and optimistic and exploratory algorithms that use recommender systems [35] to improve algorithm efficiency overtime [11, 21].

In this thesis we describe an object-level framework that supports instrumenting compiled object files to support adaptive compositional modeling. There were a number of goals we considered as important design considerations for this framework. As an object-level framework, we wanted to ensure that we were instrumenting our control functionality entirely at the object-level and did not require any modifications to the original source code. We considered the latter point important, because the source code for the application may no longer be available and any modifications will require extensive programmer time to adapt the source code, which may be represented in a multitude of different programming languages. This allows us to compose applications from collections of arbitrary codes and permits *direct code execution*, which relieves many concerns of correctness stemming from source-level modifications. We also required that the framework be able to adapt the application during runtime. With adaptivity, we can have the application explore numerous algorithmic possibilities, learn the results of these decisions, and reward the most efficient possibilities. For long running jobs on High Performance Computing (HPC) clusters, the application can typically increase its performance over time by remapping to improved algorithms for a significant portion of the time, while still being able to explore numerous computational algorithm possibilities. In other words, software systems can be configured to work in either an *exploratory* or *exploitation* mode. Additionally, attempts to create "autonomous" HPC systems will require applications to be able to adapt to the systems' availability of resources. Finally, we would like the framework to be resilient to failure when exploring solution spaces, so we would like a method to be able to "restart" the application from an arbitrary point when we detect failure. In fact, we desire to maintain the knowledge of the failure from the future so that we can use it to correct the mistakes made in the present. This allows us to iteratively apply potentially fatal algorithms to a problem until one succeeds.

To our knowledge, we believe that our approach to solving these concerns is innovative and has never been attempted. Functioning entirely at the object-level of the application requires in-depth knowledge of the machine architecture and how function call evaluations are performed. This has likely prevented many from pursuing an avenue such as this, and additionally this approach is very architecture dependent. We have implemented our approach on the x86 architecture, however the core architecture dependent kernel of our framework is encapsulated in a few small sections. Although we have not yet ported our framework to any additional architectures, the porting requirements should be localized to these sections. Many similar projects have also not considered the concept of restarting an algorithm and therefore their adaptive functionality is limited to only "safe" adaptations. Many adaptive approaches have further concentrated on making the application more aware of the environment with the use of tuning knobs, rather than leaving the application entirely unaware of the environment.

In the next chapter we begin by describing the object-level transformation and functional con-

structs we designed to achieve the goals for this framework. We continue with several simple scenarios we designed to demonstrate individual components of our framework. At the end of the chapter we present the High-level Adaptive Control Language (HACL) we constructed to allow complex adaptive decisions to be expressed easily. In chapter three we present the performance of the framework under two benchmark applications and representative usage environments. We also present several experiments in which we used our framework to "learn" the adaptivity for specific problems using reinforcement learning techniques. In chapter four we survey several works by other authors and illustrate how they relate to our framework. Finally, we present some future directions in chapter five.

Chapter 2

Basic Constructs for Adaptive Compositional Modeling

To realize our goal of a framework that can model complex adaptive scenarios, we first present a set of four primitive operations that can be used as building blocks for compositional modeling. The use of these primitives is then showcased through simple examples of designing adaptive algorithms, followed by the definition of a markup language suitable for high-level specification of adaptivity.

2.1 Primitives for Adaptive Compositional Modeling

2.1.1 Function interception

One of the basic goals of our framework is to support procedure-level decomposition of a compiled object file so that procedure calls within a module become control points at which the application's execution can be steered. Therefore, the basic construct required in the framework is a method for intercepting, or *catching*, the function calls made within an application. There are a variety of programs and projects that support intercepting function calls within an application [4, 15]. Our framework 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 which takes a single argument that is the address of the function to invoke. The framework then, for every module to be instrumented, replaces any `call` instructions with a call to our own interception handler. The original target function address is saved so that the interception handler can determine which function was originally 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 our framework, 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 our framework 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 aspect-oriented programming literature [9]) 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 our framework, 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 checks — 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. This allows the controller to massage any return values before they are passed back to the calling module to, for example, fix type differences or influence the caller.

2.1.2 Registered callbacks

When a function call is intercepted or the return of a function is intercepted, the framework should pass control of the application to a controller or online recommender system so that it may make any required adaptive control decisions. This functionality is supported in our framework using a method of registered callbacks. A recommender can register either a pre- or post- callback for a given function symbol which is executed whenever the function symbol is invoked or a return is made from the function, respectively. When the framework invokes a callback it passes a reference to the current invocation stack entry in the framework which corresponds to that function call. With the invocation entry reference, the callback can: lookup or manipulate parameters, remap the function call, search the remaining invocation stack, and checkpoint or rollback the process. In other words, the callback has full control over the current state of the application.

2.1.3 Parameter manipulation

There must be 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 gainful 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 that switching to a different routine might improve convergence.

Our framework 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 the framework, simple query functions within the framework 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 the framework is the ability to remap the entire parameter list of a function. For example, when a recommender 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 recommender specifies the size and type of the parameters of both the original function and the new function we 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.

2.1.4 Dynamic process rollback

The most valuable construct identified during the design of our adaptive scenarios was the functionality to arbitrarily 'rollback' the execution of an application to a previous state. If there are multiple algorithms that can be applied to a specific problem, but it is not known beforehand which will be successful, we can design a solver that uses an iterative, brute-force approach to solving the problem. Each iteration of the solver applies a new algorithm and, if unsuccessful, the results of the iteration are used to 'seed' the solver's decision of the algorithm to use in the next iteration. Typically, each algorithm would need to be coded to have a zero net-change on the state of the data set so that the data set is not modified between subsequent iterations of the solver. However, this requirement severely limits the flexibility in the choice of algorithms that can be used in the solver and adds considerable cost to the performance and implementation time of the algorithm. With a construct to completely revert the data set to a prior state, optimistic and exploratory algorithms can be used in the solver. Additionally, algorithms that were previously written for alternate applications can be exploited in the solver as well.

Our framework utilizes the DejaVu library, a distributed snapshot and fault tolerance library for performing dynamic process rollback. DejaVu supports online, incremental, process checkpointing of all static and dynamic memory and all file and network I/O. A function symbol can be registered with our framework to be checkpointed so that every time an invocation of that function occurs, a checkpoint is saved just prior to executing the function. Function invocations are not checkpointed by default because to do so at such a fine granularity is expensive. Instead, the recommender should determine an appropriate mix between coarse-grained and fine-grained checkpointing relative to the application. Each checkpoint operation returns a unique identifier that is stored within the respective function invocation record. This ID can be used later to rollback to the specific checkpoint.

An important criteria for a process checkpointing library used in an adaptive computing framework is that it must support the ability to maintain 'non-volatile' memory regions. If the runtime recommender decides to rollback to a previous checkpoint because the current path of execution is deemed non-beneficial, then there must be a method to ensure that the same path is not traversed again. A complete process rollback, including all static and dynamic memory, would result in an infinite loop for a deterministic application. Therefore, within the DejaVu library there is support for handling partial memory rollbacks. A dedicated memory allocator within DejaVu creates memory regions whose contents are not reverted during a rollback. Our framework takes advantage of this support by maintaining the entire function invocation stack in non-volatile memory. If

the runtime recommender decides to remap a function target or the parameters of a function, the remappings remain in memory after rolling back and are effective when the application restarts from the previous state. The runtime recommender can also register memory tuples that contain the address and size of a location in memory to maintain across rollbacks. The memory regions represented by the tuples are copied into non-volatile memory regions before a rollback and are copied back to their original locations following a rollback.

2.2 Usage Scenarios

We now present three simple use case scenarios to demonstrate the functionality supported by our framework. These scenarios bring out key elements of our framework which can then be composed in more complex settings. For each scenario, we include short sections of code that illustrate the amount of programming required in the framework.

2.2.1 Scenario 1: Switching from Quicksort to Insertionsort

A well-studied sorting algorithm in computer science is quicksort; its average case and worst case running time make it the preferred method of choice in a variety of sorting contexts. However, quicksort is also known to be inefficient at sorting small arrays of elements, or when the array is already 'almost sorted.' One common strategy is to switch to insertionsort when the length of the subinterval falls below a certain size [31]. Insertionsort is efficient at sorting small collections of elements and, unlike quicksort, is not agnostic of the 'sortedness' of the array. For instance, it is well known that insertionsort performs $O(Inv(X))$ comparisons and data moves where $Inv(X)$ is the number of inversions in X [11].

In our first scenario, we implement a naive recursive quicksort algorithm and demonstrate how we can dynamically switch to insertionsort when the array size falls below a certain threshold k (we use the term 'naive' because quicksort will typically be implemented iteratively [31] to reduce the overhead and memory requirements of recursion). How k is determined is a topic pertaining to recommender system design, and covered in the next chapter. For now, it suffices to note that the quicksort algorithm is written without any mechanism for handling the switch to insertionsort so we could model adaptation of an unmodified code module. The insertionsort routine was written so that its function prototype was exactly the same as the recursive quicksort. The following code segment indicates how the function remapping is performed:

```
void qsortcb(struct li_invoke_entry *ie, LI_HANDLER_TYPE ht)
{
        int first, last;

        /* read the arguments being passed to quicksort */
        first = LI_GET_PARAM_INT(ie, 1);
        last = LI_GET_PARAM_INT(ie, 2);

        /* if we are within the threshhold, switch to insertionsort */
        if (first < last && last - first <= k)
```

```
                li_remap(ie, isort);
}
```

To implement this scenario, we used the function interception functionality to redirect the recursive function calls to quicksort through our framework. Before the first call to quicksort, we registered a callback to intercept the call and handle the remapping using the code above. In this scenario, the callback had to retrieve the interval bounds and determine whether the interval size was less than k (a predetermined threshold). If it was, the callback remapped the call to insertionsort. To retrieve the parameters from the stack, the controlling application must specify the type of each parameter so that the framework can determine the correct memory offsets to read the parameters from. The code below is used in the initilization section, before the first call to quicksort, to specify the parameter types.

```
LI_PARAM_TYPE params[3];
struct li_symbol_entry *se;

se = li_add_symbol("quicksort", (li_invoke_func_t *)qsort, 0, 0);
params[0] = LI_PARAM_POINTER;
params[1] = LI_PARAM_INT;
params[2] = LI_PARAM_INT;

li_add_params_list(se, params, 3);
```

2.2.2 Scenario 2: DQAG Parameter Sweeping

The Quadpack [26] collection of Fortran subroutines, for solving definite univariate integrals, provides an excellent source for adaptivity scenarios. Some of the Quadpack programs (e.g., QNG) are non-adaptive and attempt to use hardwired integration rules of varying degrees of accuracy. These algorithms approximate the function to be integrated by polynomials of varying orders. Others (e.g., QAG) are adaptive, and perform recursive subdivision of the integration domain, in order to achieve specified accuracy constraints. These algorithms approximate the function using *piecewise* polynomials. Still other algorithms (e.g., QAWO) are targeted for integrals with pre-defined 'weight' functions and are primarily of the Gaussian quadrature nature, which are maximally accurate. Since all quadrature routines utilize discrete sample information, it is easy, given any integration rule, to design a problem on which the error can be unbounded [20]. When a given quadrature routine fails, there are a variety of indicators available from execution (e.g., presence of roundoff, presence of a singularity, oscillatory behavior of non-specific type) which can be utilized by a recommender to make a better judgement of which algorithm to try. The design of a completely adaptive quadrature routine, that adapts to varying integrals, would alleviate the need for the user to predetermine an appropriate quadrature routine for each integral. Such a system would be of great interest to the scientific community that utilizes these integration routines frequently.

For our second scenario we considered the double-precision adaptive quadrature routine DQAG, and attempted to determine the strictest relative error constraint (*EPSREL* parameter) it could calculate an integral to. An inability by DQAG to achieve the desired *EPSREL* would result in the output flag *IER* set not equal to zero, at which point we might attempt to lower the *EPSREL* requirement. Notice that such recovery from abnormal program terminations would be difficult if

the application modified the global state of the application, or exited upon failure, unless there was a method to checkpoint and rollback the application to the point before the routine was called. In this scenario we induce failure of DQAG by setting the *KONTROL* flag of the *XERMSG* helper routine, so that if DQAG cannot integrate the expression to the specified error bound, *XERMSG* exits with an unrecoverable error.

```
/* Flag set when XERMSG is called. */
int failed;
...
int dqagcb(struct li_invoke_entry *ie, LI_HANDLER_TYPE ht)
{

        if (ht == LI_HT_PRE)
                failed = 0;
        else {
                double *epsrel;

                epsrel = LI_GET_PARAM_POINTER(ie, 4, double);
                if (failed) {
                    /* Decrease error requirements by 10. */
                    *epsrel *= 10;

                    li_rollback(ie);
                }
        }
}
```

In the previous scenario we intercepted function calls to quicksort and read the parameters off the stack that represented the interval bounds. For this scenario, we had to actually manipulate the function parameters on the stack *before* the function was invoked. In this way, the *EPSREL* parameter could be sweeped from a low to high error until the DQAG routine eventually succeeded. Additionally, in this scenario we used the checkpointing construct to save the process state before the call to DQAG and return to it each time DQAG failed. When the checkpoint was taken, the unique identifier was stored within the invocation record for DQAG. The post-call handler for DQAG, shown above, would rollback to this checkpoint identifier to test the next value of *EPSREL*.

2.2.3 Scenario 3: Putting it all together

Our third scenario combines elements of the first and second, in that it switches from one quadrature routine (DQNG) to another (DQAG) upon DQNG's failure. In addition, the function prototypes of the two routines do not match, hence some massaging of calling parameters is required in order to perform the algorithm switching. Consider the application of DQNG on the problem:

$$\int_0^1 x^{1/2} log(x) dx = -\frac{4}{9}$$

Since DQNG does not dynamically select nodes (and weights) during the integration, it is unable to guarantee higher levels of (relative) accuracy requirements. For instance, the maximum number of function evaluations (an important metric in quadrature routines) is hardwired to 87, owing to the selection of the 87-point Gauss-Kronrod rule. When the accuracy threshold is lowered from $\epsilon_r = 10^{-2}$ in steps of 10, DQNG breaks down at $\epsilon_r = 10^{-5}$ with an *IER* set to 1. At this point, the recommender would suggest that we switch to DQAG. The following code demonstrates how the remapping is achieved:

```
struct li_param_remap remaps[14];
int i;
static int limit, lenw, last, iwork[100];
static double work[400];

for (i = 0; i < 14; i++)
        remaps[i].type = LI_PARAM_POINTER;

 memcpy(remaps[0].data, li_get_param(ie, 0), sizeof(void *));
 memcpy(remaps[1].data, li_get_param(ie, 1), sizeof(double *));
 memcpy(remaps[2].data, li_get_param(ie, 2), sizeof(double *));
 memcpy(remaps[3].data, li_get_param(ie, 3), sizeof(double *));
 memcpy(remaps[4].data, li_get_param(ie, 4), sizeof(double *));
 memcpy(remaps[5].data, li_get_param(ie, 5), sizeof(double *));
 memcpy(remaps[6].data, li_get_param(ie, 6), sizeof(double *));
 memcpy(remaps[7].data, li_get_param(ie, 7), sizeof(int *));
 memcpy(remaps[8].data, li_get_param(ie, 8), sizeof(int *));

 limit = 100;
 lenw = 4 * 100;

 ((void **)remaps[9].data)[0] = &limit;
 ((void **)remaps[10].data)[0] = &lenw;
 ((void **)remaps[11].data)[0] = &last;
 ((void **)remaps[12].data)[0] = iwork;
 ((void **)remaps[13].data)[0] = work;

 li_remap_params(ie, remaps, 14);
 li_remap(ie, (li_invoke_func_t *)dqag_);
 li_rollback(ie);
```

Many of the constructs from the first two scenarios were used to implement this scenario. We used the parameter mechanisms to read and manipulate the parameters passed to the quadrature routines. When we had determined that the quadrature routine had failed (by inspecting the pass-by-reference parameter *IER*) we had to return the process to the state it was before the quadrature routine was called. The DejaVu checkpointing library was used to checkpoint the process before the call and to rollback the process when requested. When a quadrature routine failed, it was

remapped to the next recommended quadrature routine using the function remapping construct. However, as opposed to the first scenario where we could easily remap quicksort to insertionsort because they had the exact same parameter lists, DQNG and DQAG require slightly different parameters. The code above depicts the procedures for remapping the parameters from DQNG to DQAG. Within the framework, the new parameter values are copied onto the stack in place before the execution is allowed to continue onto DQAG.

2.3 High-level Adaptive Control Language

The low-level constructs described above can be used to represent a wide range of complex adaptive scenarios. However, the constructs rely on a strong understanding of system programming and a working knowledge of the underlying code architecture. These requirements present a daunting task for researchers who are not overly knowledgeable in low-level programming but would like to write some simple adaptive scenarios to take full advantage of the framework. Even scenarios that appear trivial in functionality might require significant amount of coding. To create a more useful framework, we designed a high-level language that can express adaptive decisions easily and with relatively few constructs.

The language, termed High-level Adaptive Control Language (HACL), expresses the core functionality for adaptive decisions using an eXtended Markup Language (XML) format. The XML document representation is compiled to the low-level set of constructs described earlier using an XML parser. For the most common adaptive scenarios, users can express them using simple conditional-based action statements which require little or no code. For more complex and unique scenarios, users can include their own code which will then be embedded into the control program when the XML document is compiled. The HACL XML control document has two sections, to represent the parameter mappings and the callbacks with conditional action statements, respectively. A complete DTD for HACL is given in Appendix A.

2.3.1 Parameter Mappings Section

The parameter mapping section of the document represents the definitions of functions used in the callback section of the document. For each function, the list of parameter names and associated types must be declared in the order they are passed to the function. This allows the user to access arguments by name in the callback section without worrying about the parameter's locations in the parameter list or their correct sizes. The function definitions are also used to determine the correct parameter mappings when a user requests to remap a function. If the function has a non-void return value it can be specified as well and will be available in any post callback handlers for that function. Table 2.1 contains a sample parameter mapping for the quicksort method from the first scenario.

The parameter mapping section of the XML document can become lengthy if there are a large number of functions and would be tedious to enter manually. However, given a static source code analyzer [14] the parameter mappings could be automatically generated if the source code were available beforehand. Pre-generated or pre-defined parameter mappings could also be distributed with a project or could be queried from a database of existing mappings. Whichever method is used, the user will not have to enter numerous parameter mappings by hand.

Table 2.1: Example of parameter mappings from quicksort scenario.

```
<mappings>
  <mapping>
    <functions>
      <function name="qsort" />
      <function name="isort" />
    </functions>
    <return value="" />
    <parameters>
      <param name="a" type="int *" />
      <param name="first" type="int" />
      <param name="second" type="int" />
    </parameters>
  </mapping>
</mappings>
```

2.3.2 Callback Section

The callback section of the XML document contains the controlling constructs for adaptive decisions and allows for a lot of flexibility. Simple adaptive expressions can easily be written using XML expressions without requiring low-level system code, while more complex scenarios requiring behavior beyond the included expressions can be expressed using embedded C code in the XML document. Table 2.2 illustrates the composition of the callback section; the important sections are detailed below.

1. `<callback func="foobar" type="post" mapping="foobarmap">`

   The adaptive control decisions are specified with respect to procedure calls. With the procedure-level decomposition that our framework uses, all adaptive decisions are made when a function is invoked or when it returns. This particular line above defines the start of a callback block that is associated with the function `foobar` and is called after the function returns. The callback block header must also specify a parameter mapping that is used to determine the correct parameters for the function. For each parameter in the parameter mapping, a local variable of the same name and type is declared in the scope of the callback. The actual value of the parameter is copied from the parameter space on the stack to the local variable at the beginning of the callback. Additionally, if the mapping specifies a non-void return value and the callback is of type *post* then the local variable `_return_` will contain the return value from the function. These features simplify the further statements in the callback block since they can reference parameters by name that automatically have the correct types.

2. `<condition value="foo == 2">`

   Within a callback block there will typically be one or more condition blocks that define what actions to perform if the condition is true. The `value` attribute of the condition block is a boolean expression that, if true, continues execution of the control statements within the

Table 2.2: Example of callback section in HACL XML document.

```
<callbacks>
  <callback func="foobar" type="post" mapping="foobarmap">
    <conditions>
     <condition value="foo == 2">
       <remap mapping="func2map" func="func2">
         <assoc alloc="exist" from="foo" to="bar" />
         <assoc alloc="stack" name="newvar" type="int" to="foo"
           value="100"  />
         <assoc alloc="dynamic|global|existing" />
       </remap>
       <rollback search="search_func" data="..." />
       <code type="pre|post">
         ...
       </code>
     </condition>
     <code type="pre|post">
       ...
     </code>
    </conditions>
  </callback>
</callbacks>
```

condition block. Each condition block's boolean expression is tested sequentially in the order they are specified and the first one that evaluates to true is executed. Currently, only one condition block is evaluated for a callback.

3.
```
<remap mapping="func2map" func="func2">
  <assoc alloc="exist" from="foo" to="bar" />
  <assoc alloc="stack" name="newvar" type="int" to="foo"
    value="100"  />
  <assoc alloc="dynamic|global|existing" />
</remap>
```

   Each condition block has an associated set of action statements that specify what to do in case the condition is true. We have identified that a majority of the adaptive decisions involve remapping a function and/or rolling back the function to a previous state. Therefore, we have defined statements for easily performing these operations. The statements above represent how function and parameter remapping is handled. For this specific condition, the function is to be remapped to `func2` using the new mapping `func2map`. The new function mapping must have been already declared in the mappings section (see section 2.3.1) of the XML document. For the parameter remapping, the associations from the old to the new parameters must be included; if the parameters are the same, no association needs to be included. For each parameter of the new mapping, an `assoc` statement must be included. An *existing* association maps a parameter from the current mapping (specified in the `callback` block header) to a parameter of the new mapping. An association can also declare a new variable (either as a *global* variable or as a *dynamic* variable) and assign it a new value and use it as the new parameter.

4.
```
<rollback search="search_func" data="..." />
```

   If included, this statement will signal the framework to rollback the function to the associated checkpoint location. The *search* field specifies how to locate the checkpoint location to return to as a search function. The search function is passed a handle to each invocation entry in the framework, along with the user data, and should return true when it locates the location to rollback to. There are several predefined search functions that locate, for example, the current invocation entry, the first called invocation entry matching the given function name, and the last called invocation entry matching a particular function name.

5.
```
<code type="pre|post">
  ...
</code>
```

   Code sections can be embedded into a condition as well. If the user requires operations that are more complex than simply remapping and/or rolling back a function, they can embed a code section into the condition. The code section has full access to all the parameters from the current mapping and can modify these values before they are remapped or saved back to the arguments on the stack. Of course, remappings and rollback operations can be performed manually in the code blocks as well.

Chapter 3

Framework Performance Evaluation

The framework discussed in this thesis encapsulates a range of functionality for realizing various adaptive modeling problems. With this functionality comes a measurable performance penalty that must be incurred to exploit the constructs. At the most basic level, the framework behaves as a function interception library that redirects the function calls made in a code module through the framework. To determine what target procedure is being invoked when a redirection occurs, the framework has to lookup the procedure based upon the four-byte function address. To map between the function address and a symbol entry structure, the framework uses a hash table structure. The hash function used to hash the address of the invoked procedure is the Bob Jenkins hash function [16]. Jenkins' hash function is relatively fast, on the order of $O(6n+35)$, and has a small probability of collisions, about one in every 2^{32}. However, despite the efficiency of the hashing function, the library incurs this penalty for every function call instruction executed in the code module and therefore is proportional to the number of function calls made. Table 3.1 illustrates the approximate base runtime cost for executing a redirected empty function. To reduce the performance penalty, the user of the framework can selectively pick which modules they would like to instrument with the framework instead of automatically redirecting function calls from *all* the modules within a program.

While the direct performance overhead from intercepting function calls can be significant, the advantages gained from being able to dynamically switch algorithms for optimal performance can outweigh the performance costs. For example, the solution space can be explored using data mining and reinforcement learning [33] approaches to determine the optimal algorithm to use for given input values. By mining the solution space for a better approach to a given problem [27], the framework can improve the observed performance despite the overhead of using the framework's function redirection.

Table 3.1: Comparison of execution times for an empty function.

Execution Method	Execution time
No Framework	0.006 μs
Function Interception	0.103 μs
F. I. and Callback	0.184 μs

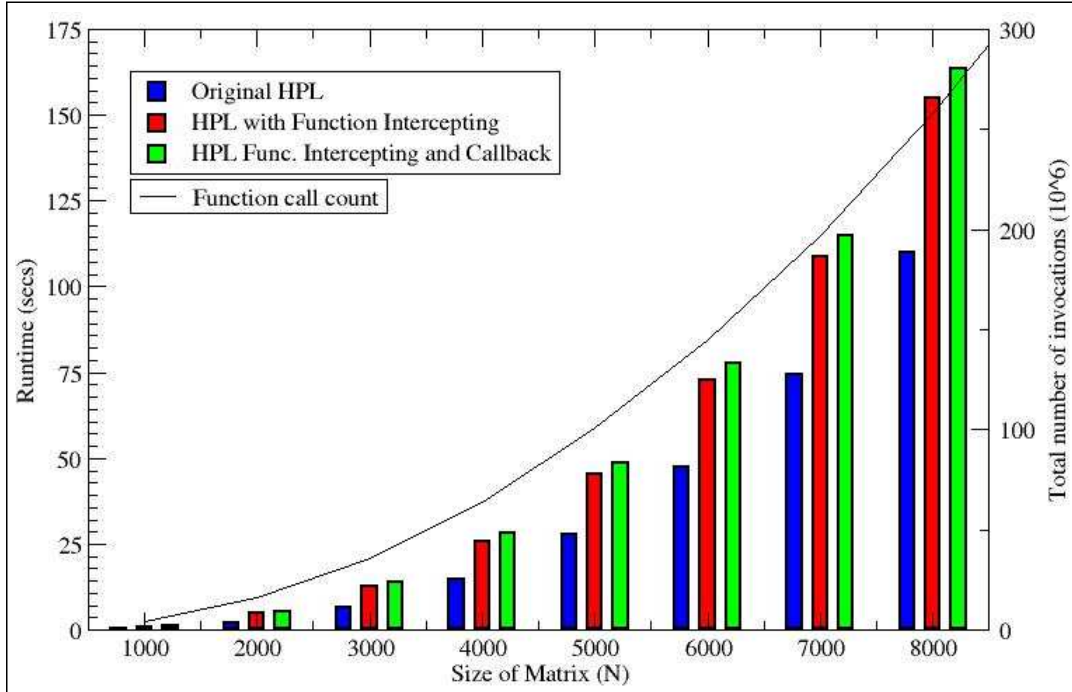

Figure 3.1: HPL Performance with Framework.

3.1 Benchmarking

To demonstrate the performance of the framework, we instrumented two key benchmark applications with the framework and measured the runtime increases compared with the original uninstrumented benchmarks. The first application we employed was the High Performance Linpack (HPL) benchmark application that solves a system of equations on distributed-memory systems. The tests were run on a single 2GHz AMD Athlon64 processor with 1GB of memory. The results, as seen in Fig. 3.1, were calculated for varying sizes of the linear system. The HPL benchmark application makes an enormous number of function calls throughout its execution and the difference in runtimes between the original HPL and the instrumented illustrates this aspect. For the test with $N = 8000$, the number of function calls is more than $250,000,000$. The results of the test also demonstrate that adding a registered callback for a function symbol did not add any significant additional overhead compared to simply instrumenting the code for function redirection. In the HPL test we added a callback for the function `HPL_lmul` which was recorded as receiving the most invocations during a single run of the benchmark.

In comparison, we also instrumented our framework for the ASCI Sweep3D benchmark. The Sweep3D benchmark does not have a large number of function calls, only on the order of about 500 per test, and therefore the performance overhead was insignificant as pictured in Fig. 3.2.

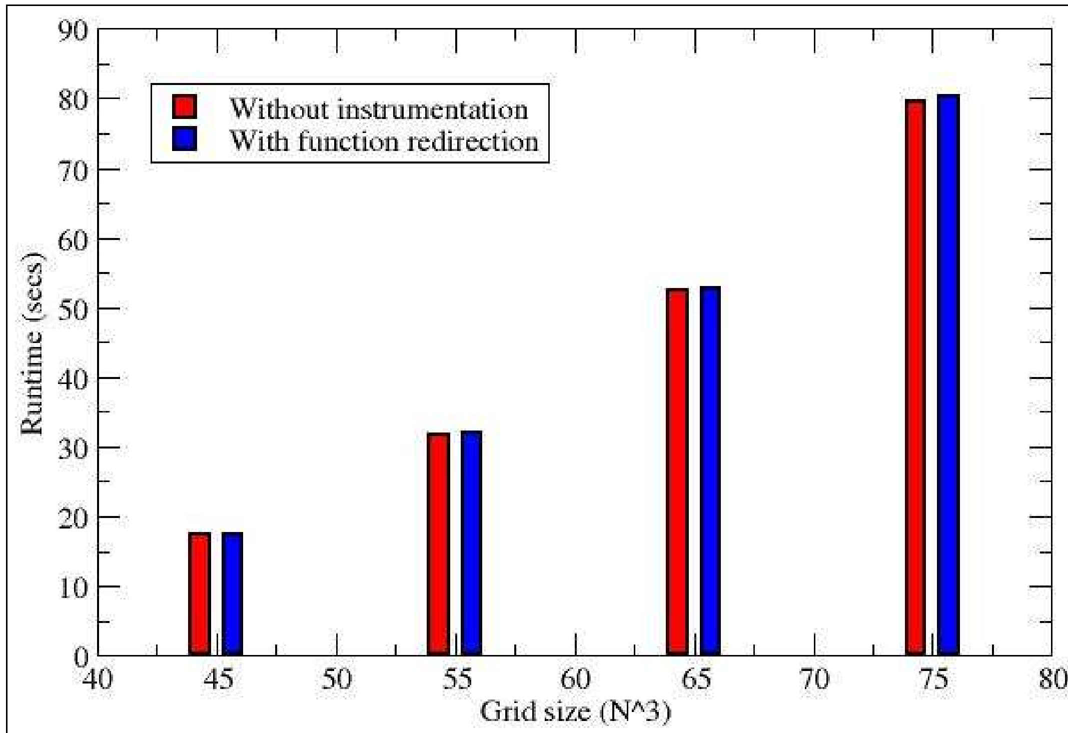

Figure 3.2: Sweep3D Performance with Framework.

3.2 Learning to be Adaptive

In this section we describe how the framework can be embedded in a recommender system context, for realizing adaptive algorithm selection applications. The design of recommender systems is a major topic by itself, and outside the scope of this thesis. For more information, see [5, 35].

By using the framework we could manipulate the codes to alter their execution dynamically, so that they explore alternative solutions and eventually exploit these alternatives to optimize the algorithm. The choice of which algorithm to use at each step is encapsulated in the form of a probabilistic *policy* (that apportions a probability mass of 1 among all applicable algorithms). One way to learn such policies is through Monte Carlo estimation.

In MC estimation, different runs are made with different algorithmic choices, in an effort to estimate relative efficiencies and assess selective superiorities. At each run of an algorithm that involves multiple decision points, the choices made at each decision point are recorded and results are amortized to calculate the efficiency and performance of each run. Several reinforcement learning techniques are available that learn to reward choices that resulted in the most desirable execution outcomes. The rewards were associated with specific input criteria so that algorithms could provide comparable efficiency for varying degrees of input. Our results from experiments with adaptive sorting and adaptive quadrature are presented in the next sections.

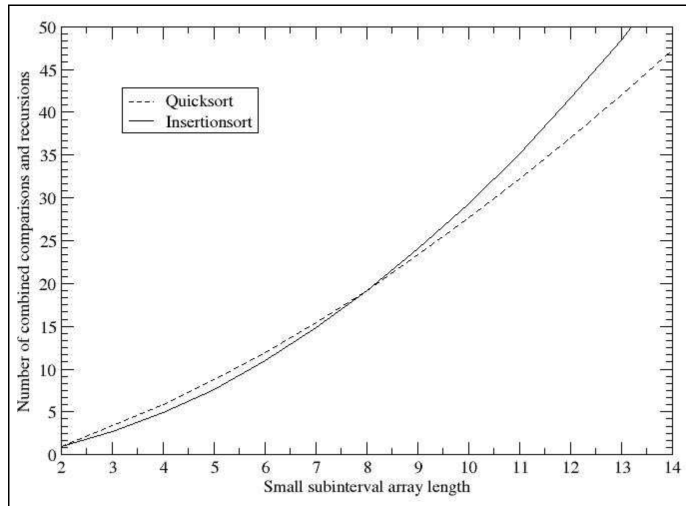

Figure 3.3: Comparison of total operations between quicksort and insertionsort.

3.2.1 Sorting using Two Algorithms

In our first scenario we instrumented a recursive quicksort routine, intercepted the recursive calls, and remapped them to insertionsort when the array subinterval got smaller. Here, our goal is to measure the efficiency of quicksort and insertionsort so that we could determine an optimal cutoff point to switch from quicksort to insertionsort. We begin with a random policy that encourages 'exploration,' i.e., at each decision point, either algorithm has an equal probability of being selected. The callback handler thus effectively randomly decides whether to switch quicksort to insertionsort at each subinterval. When the subinterval sort was completed, the efficiency of that specific sort was calculated and recorded. With each such run, the policy is slightly 'nudged' toward the algorithm that performed better, always retaining some probability mass for the other algorithm (to facilitate exploration).

To measure efficiency in this problem we experimented with a variety of problem characteristics. The number of swap operations and the number of comparisons that each algorithm performed was recorded for each subinterval when sorting an array of randomly distributed elements. As was expected, quicksort performed significantly fewer comparisons and swaps at larger array sizes than insertionsort. At array sizes less than five elements, quicksort and insertionsort had similar efficiencies for the number of comparisons and swaps. We then added the number of recursions into the efficiency along with comparisons, and this produced some interesting results. For the smaller subinterval sizes, quicksort's high recursion rate made it less desirable than insertionsort. As illustrated in Fig. 3.3, there is a cutoff point at eight elements where insertionsort requires fewer operations. This cutoff point, between eight and nine elements, is one less than the cutoff point suggested by Sedgewick in his seminal paper on quicksort implementations [31]. In fact, given that a single recursion operation is more costly than a comparison, the true cutoff would be higher if a recursion were given more weight than a comparison.

3.2.2 Designing an offline recommender for DQAG

Recall that in the second and third scenarios, we experimented with several different quadrature algorithms to determine the most precise error bound an integrand could be evaluated to using a given quadrature routine. We also created an automatic adaptive quadrature system to switch between quadrature routines. A criteria that is important to consider when using an adaptive quadrature routine is the number of function evaluations required to achieve the requested accuracy. Function evaluations can typically be rather expensive depending upon the complexity of the expression being integrated; therefore, unnecessary evaluations should be avoided whenever possible. The adaptive quadrature routines provide a *KEY* parameter that specifies how many function evaluations to use on each subinterval the adaptive routines divide the integral into. The same number of evaluations are used on each individual subinterval regardless of the characteristics of the specific subinterval. If the caller of the routine specifies to use a 15-point quadrature rule, an interval that only requires 20 evaluations to achieve the requested accuracy will end up taking an initial 15-point evaluation followed by two additional 15-point evaluations in each half of the interval, resulting in 45 total evaluations. If a recommender could, over time, learn the characteristics of the specific problem it could realize that a single 21-point evaluation of the same interval would be sufficient. We focused our attention on the oscillatory integrand:

$$\int_0^{\pi} cos(2^{\alpha} sin(x))\, dx \tag{3.1}$$

As was mentioned, we would like to design a system that can explore the solution space of a problem to learn better, more efficient, algorithm choices. We first focused on designing an *offline* recommender system that organizes a performance database of runs and then mines the database to identify promising algorithmic choices. In the next section, we outline how an *online* recommender system is built that interweaves data collection and algorithm assessment.

In this experiment, the framework was used to instrument the quadrature routine DQAG so that we could dynamically alter the parameter that specifies the number of point evaluations to use. Over a series of invocations, the parameter was randomly chosen among a range of six values that select the number of point evaluations between 15-61. When the DQAG routine completed, we intercepted the function return and recorded the number of function evaluations — failure of the DQAG routine to integrate the expression to the correct accuracy was also recorded. The value of α in the integral was also randomly chosen, along with the requested relative error threshold ϵ.

From the viewpoint of the recommender, we defined a state s as the 2-tuple (α, ϵ) and defined the action a to be the discrete choice of quadrature rule to use (influencing the basic # point evaluations). Therefore, the pair (s, a) defines a point in a specific episode for which we recorded a cost R equivalent to the number of resulting function evaluations. The choice of a given a state s was chosen at random until all a's had been tested for state s. At this point we used an on-policy Monte Carlo control algorithm [33] to choose the optimal a for given state s. For each returned cost R we computed the average of all R's over the point (s, a) and stored this as the q-value $Q(s, a)$. The algorithm for computing the desirability of each action a in state s is shown in Eq. 3.2. The value a^* is computed to be the cost of the best choice of a in state s and ensures that the best choice gets the highest chance of selection. The 'on-policy' characteristic of the MC algorithm means that the policy that is being learned is also the policy that is used to generate random episodes. To ensure the validity of on-policy learning, the policy must ensure that the best choice for a will be taken a

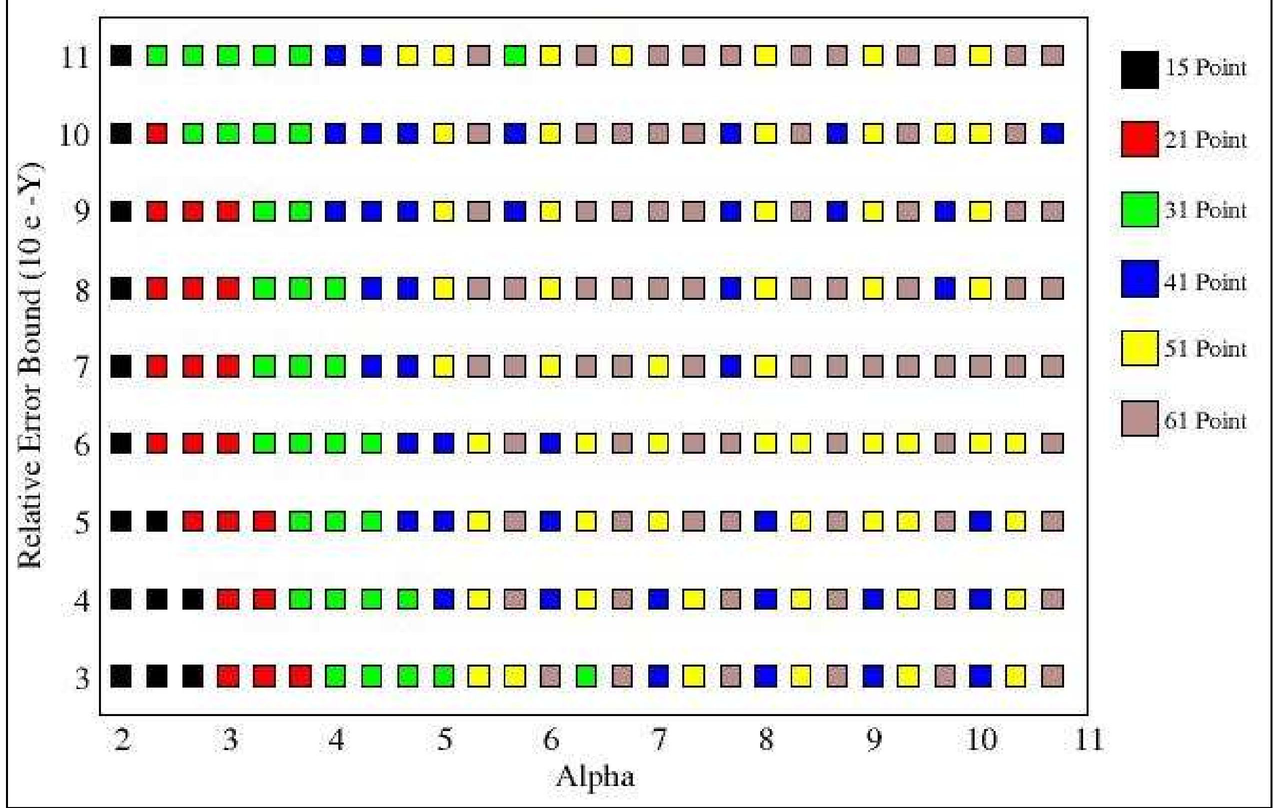

Figure 3.4: Color plot showing distribution of optimal *KEY* values for DQAG usage.

majority of the time, but that non-optimal choices will still be selected some portion of the time. This ensures constant exploration.

$$\pi(s,a) \leftarrow \begin{cases} 1-\epsilon+\frac{\epsilon}{|A(s)|} & \text{if } a = a^* \\ \frac{\epsilon}{|A(s)|} & \text{if } a \neq a^* \end{cases} \tag{3.2}$$

After running a test program that attempted to integrate expression 3.1 50,000 times, we were able to learn accurate policy estimates for choosing each a for a given s. Fig. 3.4 is a color plot illustrating the optimal a for each s. The diagram demonstrates that for smaller values of α and ϵ, a lower-accuracy point-evaluation rule is sufficient to accurately integrate the expression. However, as α increases the integral requires more evaluations to correctly integrate. Similarly, as the requested accuracy ϵ increases, the problem requires more point evaluations. This shows that the lower-accuracy rules fall out of favor as we move toward the right (and top) of Fig. 3.4. However, these portions of the graph are significantly more noisy and demonstrate that there is not a monotonic relationship between *KEY* and α or ϵ.

This experiment is rather naive, as the only utilization of the framework was in catching the call to the integrand function to add the extra parameter α, and the call made to DQAGE to manipulate the *KEY* and *EPSREL* parameters. While both of these operations could have been implemented by refactoring at the source-level, the purpose of this experiment was to demonstrate that a recommender system could use the framework to easily explore multiple algorithm parameters to assess

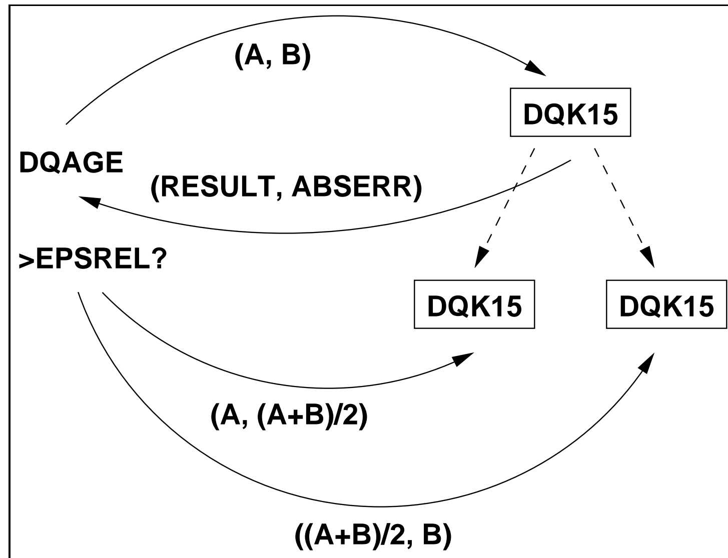

Figure 3.5: Example illustration of the operation of DQAG(E).

optimal performance for given inputs. The system could also then use the framework to exploit this gained knowledge at runtime to dynamically alter the algorithm to achieve optimal performance. The next example demonstrates this capability.

3.2.3 Designing an online recommender for DQAG

In the previous section we designed an *offline* recommender system that learned the optimal *KEY* value to use for each value of the problem characteristic α and the error bound ϵ. The results acquired from the experiment could be stored in a database or they could be used as a reference for setting the *KEY* value when attempting to integrate Eq. 3.1 with set values of α and ϵ. In this section we build a system that determines an adaptive policy automatically and uses this policy to efficiently integrate the expression, without knowledge of α.

The globally adaptive characteristic of DQAG implies that it will continuously subdivide the interval with the greatest error, and consider all intervals at all times, as candidates for further subdivisions. This recursive process is illustrated in Fig. 3.5, even though as implemented the algorithm is not recursive. DQAG will first evaluate the function with the given range using the quadrature rule specified by the user. The quadrature rule returns the estimate (*RESULT*) of the integral (*I*) in the given range and an estimate (*ABSERR*) of the error (*ABS(I - RESULT)*). If the error is greater than the requested accuracy, DQAG will bisect the interval and evaluate each half. Each subinterval that is evaluated is added to a list of subintervals which make up the result across the entire interval. DQAG will sort this list by decreasing *ABSERR* and will continuously subdivide the interval with the highest error until the problem has been solved to the requested accuracy or the maximum number of evaluations has occurred. The way DQAG is designed, it will keep reusing the same quadrature rule on each subinterval regardless of any other information that might be available to it. Fortunately it doesn't matter to DQAG exactly what quadrature rule is used to evaluate each subinterval, as long as it returns a result and an absolute error, and the number of

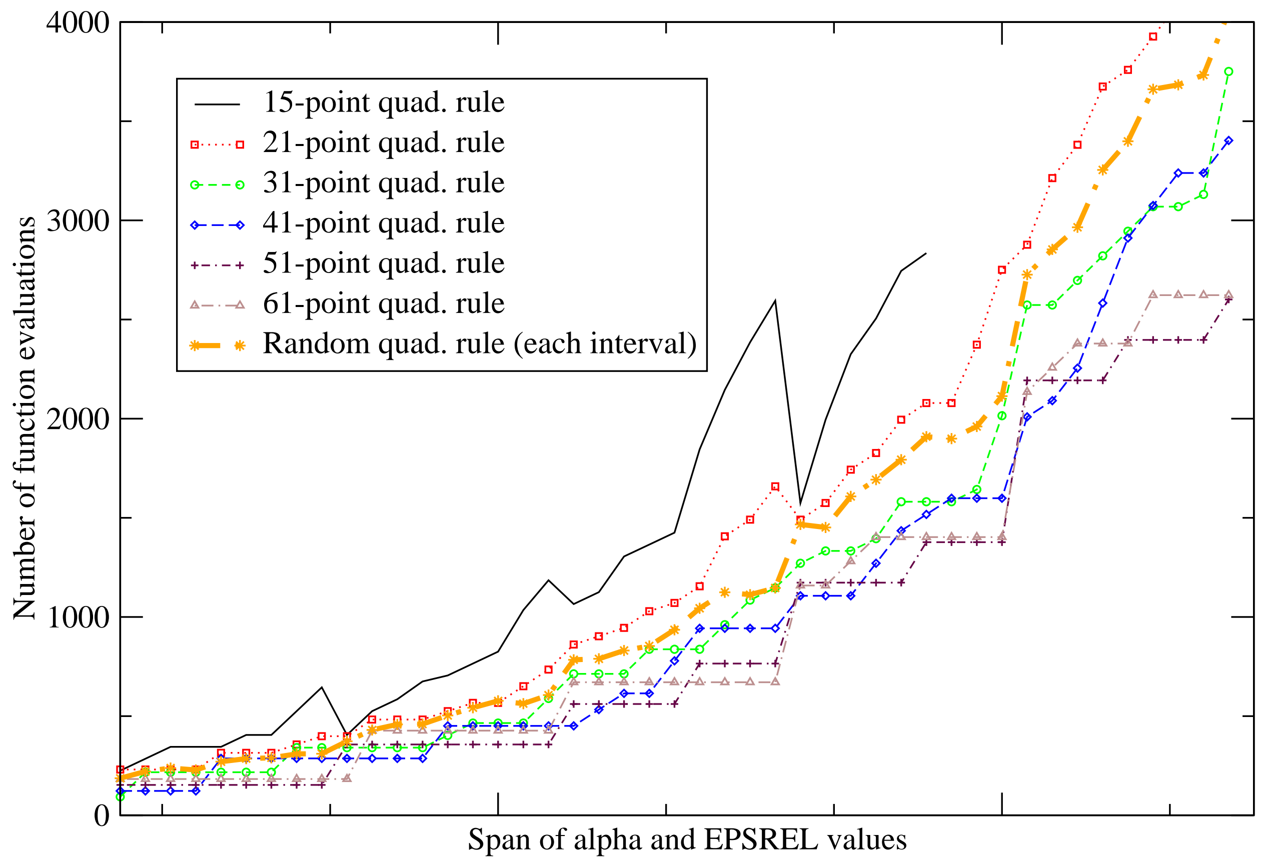

Figure 3.6: Comparison of fixed point quadrature rules and random policy.

point evaluations is monotonically increasing for later bisections of a given interval. Therefore, we can remap the function calls made to the quadrature rules to dynamically adjust the number of point evaluations depending upon the state of the globally adaptive quadrature routine.

The first scenario we tested was to randomly pick a quadrature rule each time we intercepted one. Using the six default quadrature rules (15-, 21-, 31-, 41-, 51-, and 61-point) one was selected randomly and the original function call was remapped to it. This was very simple to perform as all six quadrature rules take the exact same parameters, so no parameter remapping was required. The DQAG routine was called 20,000 times on the expression 3.1 and each time the α and ϵ were randomly chosen. On return from DQAG, the number of function evaluations were recorded for the given α and ϵ; if DQAG failed to integrate within the maximum number of evaluations, no result was recorded. As was expected, the number of function evaluations experienced for the randomly chosen quadrature rule policy was about equivalent to averaging the number of evaluations used by the fixed key policies, as is illustrated in Fig. 3.6. This shows that even a policy as naive as randomly altering the quadrature rule can perform better than selecting a single quadrature rule to use across all values of α and ϵ. However, this policy does not use any state of the problem in determining its policy. We should be able to design a smarter system.

The next policy we tested used the increasing depth of the interval subdivisions as a factor in the problem. As shown in Fig. 3.5, DQAG will continuously "recurse" (in a sense, it is actually

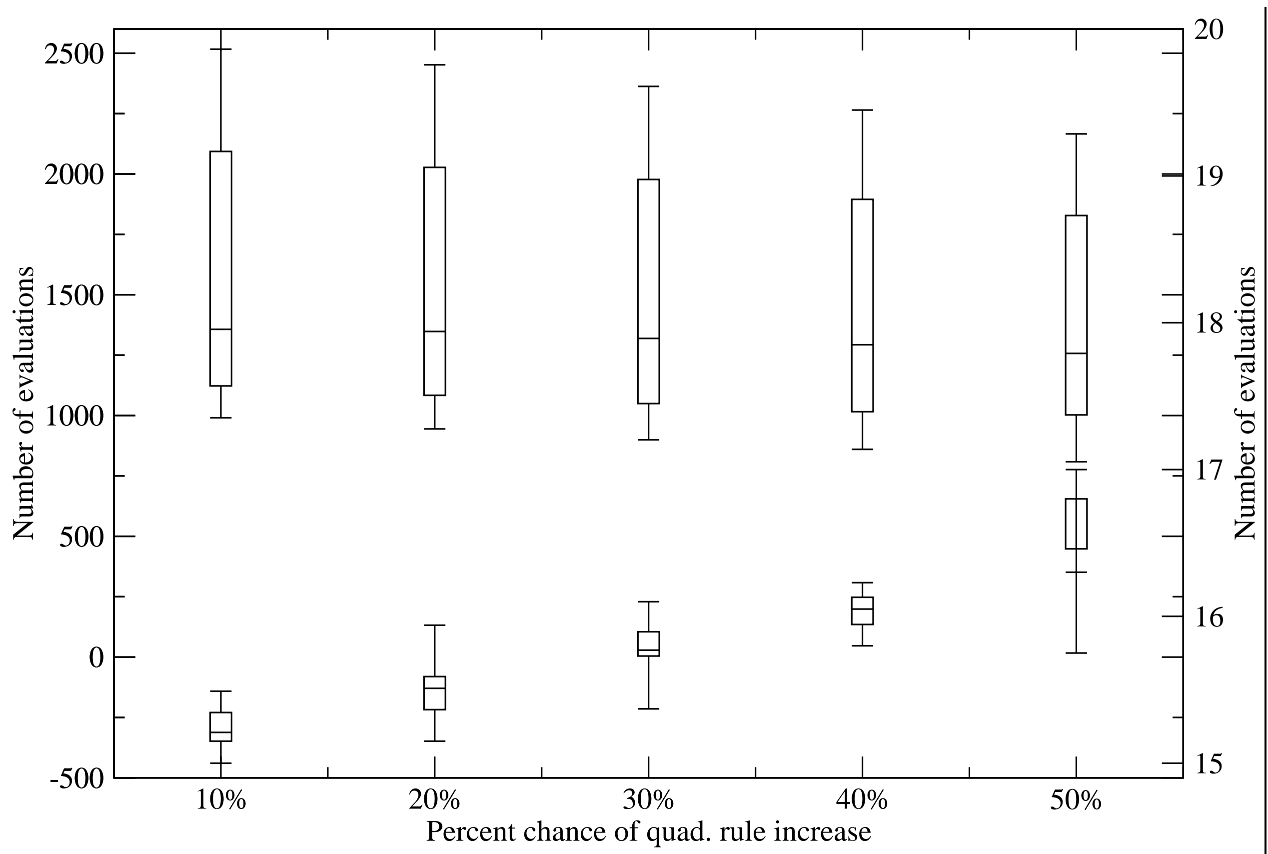

Figure 3.7: Plot of the number of required evaluations for differing probabilities of quadrature rule increase. The boxes on the top represent the number of evaluations (measured on the left y-axis) for large values of α in expression 3.1; the boxes on the bottom represent the number of evaluations (measured on the right y-axis) for small α's.

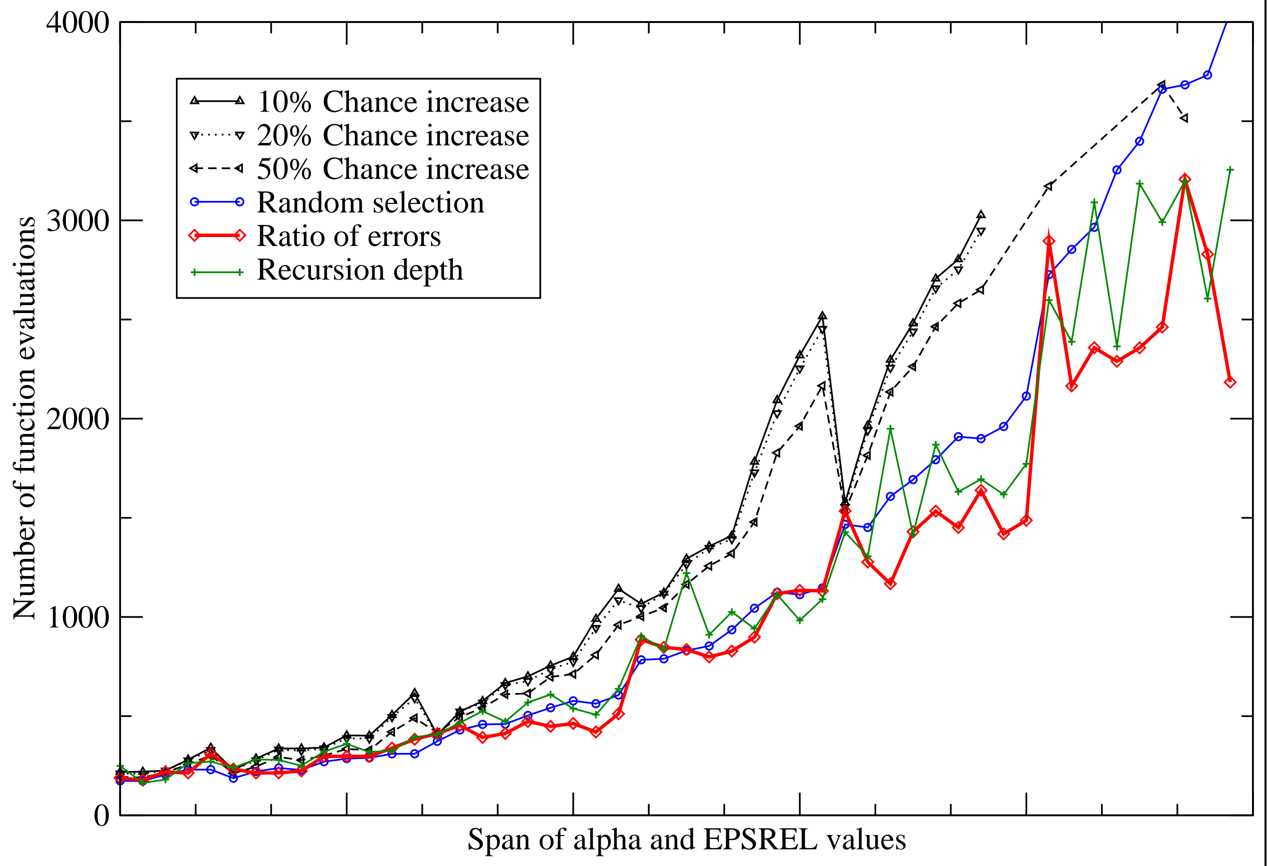

Figure 3.8: Comparison of all the RL policies we tested.

iterative) on an interval if the error is high, which usually implies that there is a troublesome behavior in that region of the integrand. Therefore, instead of continuing to subdivide the interval and using the same quadrature rule on each subinterval, it might be more beneficial to use a higher point quadrature rule. In this experiment we start with the lowest point quadrature rule (15) and for each subsequent recursion we increase the *KEY* of the quadrature rule by one, based on a predefined probability. The probabilities we initially tried were 10%, 20%, 30%, 40%, and 50%. Figure 3.7 graphs the probabilities along the x-axis and the number of evaluations between 15-17 used across all values of α and ϵ along the right y-axis using a box plot. The increasing boxes on the bottom of the graph illustrate that, for low values of α and ϵ, where the number of evaluations are low, a policy that increases the quadrature rule 10% of the time is the most efficient, whereas increasing the quadrature rule 50% of the time is less efficient. This is because the problem is fairly simple to solve at these values of α and ϵ and therefore a low-point quadrature rule is sufficient, but a high-point quadrature rule makes unnecessary evaluations. The contrast is shown in the top of Fig. 3.7 which shows the higher evaluations range for the same problem. Here, the lower percentage of increase policy results in higher evaluations because it must perform more subdivisions. The results from this test indicate that a "smart" policy could be developed that used an approximation of how difficult the problem is to determine when to use a higher point quadrature rule. However, the value of α is not known beforehand, and we must design a surrogate representation of "state".

We chose two methods for determining the difficulty of the problem and used them to decide whether to increase the quadrature rule. The first method we used was the ratio of the error resulting from the current subdivision to that of its parent's error. This ratio signifies how much of the error from its parent the current subdivision received. If the subdivision received a high percentage of the error, it will potentially want to increase the quadrature rule it is using; whereas the other subdivision that received a small portion of the error, can continue to use the same quadrature rule since it will likely not be subdivided. We defined four actions for this problem: *SS* (both subdivisions remain using the same quadrature rule as their parent), *SI* (the left subdivision remains the same and the right increases its quadrature rule), *IS* (the left subdivision increases and the right remains the same), and *II* (both subdivisions increase their quadrature rule). For this problem we used the same MC reinforcement learning algorithm as in Eq. 3.2 to reward the policy that resulted in the lowest number of function evaluations. The rewards were associated with the error requirements that were specified to the DQAG call, the range of the current error to parent error, and the policy that was used to obtain those rewards. Note that in this experiment we did not correlate the rewards with the value of α because we were designing a recommender system that should behave based upon the state of the inputs and not with regard to the difficulty of the problem. This was the key point of this experiment: we wanted to learn a generic policy that could be applied without prior knowledge of the problem difficulty, α, and which would still provide reasonably efficient results. Figure 3.8 shows the results of all the different approaches we have tried.

The second method we chose to determine the difficulty of the problem was the depth of the subdivisions along the current branch. The results from this policy are also plotted in Fig. 3.8 and show that even this naive method performs better than our previous policies. However, the method of using the error ratios still performs better in most cases.

In this experiment we had chosen two simple methods for approximating the difficulty of the problem. These approximates were values that were simple and computationally quick to compute on-the-fly during the execution of the algorithm. However, despite their simplicity, our results show that they are still useful as determinants for the current difficulty of the problem. More complex methods could be easily be developed that utilized a wider spectrum of knowledge gained from the execution to predict the current difficulty of the problem. Additionally, more actions could be defined than the four (*SS*, *SI*, *IS*, *II*) we used in this experiment.

The assumption in this experiment was that we were working with only one type of integral. The integral in expression 3.1 is an oscillatory function with problem difficulty increasing with the value of α. Therefore, the adaptive policy that we learned during the runtime of the experiment was only directly applicable to integrals drawn from a similar family of problems. If we used the learned policy on a different integral, we would likely find that it does not perform very efficiently. To use the approaches in this experiment we would have to add an additional variable that correlated the characteristics of the problem with the results of the learned algorithm. Such a characteristic could be determined with similar methods that were used in this experiment. The procedure of mapping a problem characteristic onto a learned policy is an active area of current research [28].

Chapter 4

Related Work

This chapter presents related work in runtime frameworks for compositional modeling.

4.1 Post-Object Programming Mechanisms

Object-oriented programming has proved itself to be a very useful and successful programming paradigm. The ability to modularize components of the system into separate, self-contained objects that can be maintained individually, and designed to adhere to individual contractual requirements, has been successful in many projects. There have been numerous objected-oriented languages developed along with many object-oriented tools and methodologies. However, despite its success, there are some limitations to the OO model. Many OO projects have conflicting design factors that the project can be decomposed by, but in the OO model only one of these factors can be chosen. The remaining factors, or concerns, can end up spread across multiple classes throughout the project. This is a common problem with project-wide constraints like security and quality of service [9] or system-level requirements. If the project is decomposed depending on the major requirements, the functionality to handle the remaining concerns can become dispersed across multiple classes where it is difficult to locate and maintain. To address some of these issues, there has developed a variety of Post-Object Programming (POP) methods, including Aspect-Oriented Programming and Generic Programming.

4.1.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) takes the view-point that programs should be specified as a set of concerns, or aspects, and by how these aspects should interact. AOP addresses the "cross-cutting" problem with object-oriented languages that occurs when concerns are spread across multiple classes. AOP takes the multiple overlapping concerns and attempts to weave them together. The individual concerns are organized as modular elements of the system that can be separately developed; this avoids the substantial work required to change a system concern that is intertwined among multiple classes of the project. When the collection of concerns is weaved together to compose a single coherent program, the AOP model creates abstract "join points" to connect the individual concerns. These join points are meant to be abstract communication points between the modules and are usually represented as subprogram invocations. Several adaptive program-

ming techniques [18, 22] have been developed to address the need for adaptive join points between modules, so that the invoking subprogram does not have to know the correct semantics for performing the invocation. Our framework allows a user to achieve similar goals by allowing them to remap function invocations and arbitrarily manipulate function parameters. Furthermore, since code modules can be modified without recompiling, pre-existing object files can be loosely combined to interact as different concerns of the system.

4.1.2 Generic Programming

Generic programming represents developing a program by focusing on the concepts of the program rather than on the actual code. Concepts are modularized into specific abstractions that meet a given set of basic requirements and that perform the required program concept. Programming using concepts involves determining groups of requirements that can collectively be implemented with a certain degree of abstraction, yet still function efficiently on the individual components [24]. The sets of requirements are modeled as concepts which are implemented generically and which can be applied to multiple families of components. A successful and popular example of the generic programming model is the C++ Standard Template Library (STL) [32]. The STL contains a collection of algorithms that follow common guidelines for functionality across the wide range of the STL's container and iterator classes. For example, the *sort* routine within the STL is implemented generically so that it can be used with any container class that supports iterators, while still providing an efficient sorting routine. Part of the difficulty in implementing an algorithm that is both efficient but yet abstract, is handling the range of different inputs, some of which may cause the algorithm to approach its worst case time bound. David Musser discussed a method for improving quicksort [25] when it reaches its worst case operating scenario by dynamically switching the algorithm to one with a better worst case time bound. The median-of-3 quicksort algorithm commonly implemented in STL *sort* approaches quadratic running time when the partition sizes are drastically uneven. Musser's *introspective* sort dynamically switches to heapsort when the recursion depth of quicksort approaches a certain logarithmic bound. This concept is similar to the study presented earlier which tried to determine how to optimally switch from quicksort to insertion sort, to reduce the number of comparisons and recursions. Our framework can be used to empirically determine the optimal points to switch an algorithm to improve performance with certain given inputs without losing any abstraction of the algorithm. The framework can also be used to perform the dynamic algorithm switches without having to rewrite the algorithm to handle the switching.

4.2 Tunability Interfaces

There are a number [7, 12, 17, 23, 29, 37] of approaches for specifying adaptive decisions an application can perform to adhere to performance requirements under varying resource availability. In these frameworks, the application will specify the critical resources it depends on and how to handle degradation of these resources. A performance monitor tracks the availability of resources within the system and alerts a control module when a resource has reached a critical level. The control module can invoke routines within the application that adapt, or tune, the application to the current resource availability by changing the perceived quality of the application or utilizing

different algorithms within the application. A common example [12] is streaming a video over a network. If the bandwidth degrades, the application can be alerted, which will switch the video stream to a lower frame rate or higher compression algorithm, at the expense of video quality and CPU usage, respectively.

Our approach is not necessarily targeted at resource tunability; however, the generic adaptation controls our framework supports allow applications to be tuned so they maintain specific performance requirements under varying resource availability. A resource monitor could track resource availability within the system and, using the function remapping constructs, remap specific functions like compression routines to dynamically adjust to the resource availability. Our approach can be applied to compiled object files which does not require rewriting the application to interface with a tuning API. The framework by Chang and Karamcheti [8] allows specification of resource constraints and adaptation controls in an external tunability interface. Their framework uses a similar approach of API interception, which allows them to adapt the application without having to rewrite the application. API calls to operating system resource functions can track resource availability and usage and can adapt the application if requested.

4.3 Compositional Modeling

Our framework is similar to the literature on compositional modeling. Compositional modeling involves modeling the system as a collection of distributed executable components. The question that compositional modeling literature addresses is how to compose these components, either statically or dynamically at run-time, into an efficient system configuration model. Often, these configurations are constructed by running simulations, performing empirical measurements, or by human and computer analysis of the application and its execution environment. This methodology is known as Model-Based Control [1] and it produces effective initial system configurations. In adaptive model-based control, the system can be adapted to run-time changes that affect the performance of the application.

The Performance Oriented End-to-end Modeling System (POEMS) [2], is a framework for constructing performance models for parallel and distributed applications. The POEMS framework permits the composition of a multiple domain model, including analytical results, simulation data, and direct measurements, into a single system model. It also permits components written in different languages and on different architectures to be composed into a single multiparadigm component model. POEMS represents these multidomain, multiparadigm models using expressive task graph models. The task graph models are manipulated in correlation with performance measurements obtained from the execution environment. To abstractly connect the components in the system, the POEMS Specification Language (PSL) interface is used. The PSL is a common language interface that components export which express the mappings between components. Our framework, along with the Weaves Reconfigurable Programming Framework, can be used as the computational substrate needed to connect the components in the system. Manipulations to the task graph in response to obtained performance models of the execution environment can be done using the function and parameter remapping constructs of our framework.

The Program Control Language (PCL) [10] defines a high-level control language for expressing adaptation operations of an application. PCL uses a static task graph representation of a program, similar to the POEMS framework. The PCL framework requires instrumenting the target program

with PCL operations, but our framework can be used with PCL to avoid these changes. The original locations in the target program that were previously instrumented with PCL routines can be intercepted in our framework and exported as PCL control points.

4.4 Binary Interception

Debuggers and code profilers have long used binary rewriting techniques to manipulate programs in order to set and catch breakpoints or record profiling information. These methods do not require the application source code to be modified. The binary patching can occur at compile time, when it is easy to patch the high-level assembly instructions, or it can be done after the program has been compiled to an executable object. However, the latter option requires careful analysis of the entire executable object so that offsets and alignments are preserved.

There are at least two related approaches that target the binary interception of functions calls within a program. The Mediating Connectors framework [4] uses a technique of overwriting the PLT of a shared library to intercept function calls. When an application makes a shared library call, the function is looked up in the PLT to determine its location in the shared library. If the entries in the PLT are overwritten, the calls from the application to the shared library functions can invoke wrapper functions within the Mediating Connectors library instead of the original shared library functions.

Another binary interception technique, used in the Detours [15] project, involves rewriting the beginning of the target function to make a call back to a handler function. This requires extracting the first few instructions of the target function at run-time and replacing them with an instruction to invoke the handling, or "detour", function. The extracted instructions are added to a trampoline function which can be invoked if the user requests. Detours can also be used to edit the link table for DLL's, similar to how the Mediating Connectors framework edits the PLT. Our framework is similar to the Detours project, except that instead of intercepting function calls at the beginning of the target function, we intercept them at the point the call is being made. Our framework patches the caller to call a handling function instead of the target function. When a call to an intercepted function is made, the user can decide whether to continue execution with the original target function or to continue with a different function. With caller-side patching, we do not have to have the target function available at run-time to patch. We can use "very-late" binding and load the function when it is first executed and intercepted.

Chapter 5

Discussion

In this thesis we have described a runtime framework that facilitates adaptive compositional modeling and we have illustrated its use in a range of applications. The framework was originally designed with the goal in mind that one should be able to compose an application from different arbitrary scientific codes and harness the ability to spatially and temporally adapt the application across the selection of codes and execution paths. We also designed the framework so that we could instrument this functionality without having to modify the original source code to interact with any specific API; in fact, we envisioned the possibility that the source code might not even be available at all and that we should implement our framework to function at the binary object level. We have actually illustrated throughout this thesis that the real role of the framework is two-fold.

The first role of this framework is in defining the adaptive policies that an application should make use of during its execution. Using the basic construct of intercepting individual function calls, the application can represent adaptive decisions as manipulations to the procedure-level decomposition of the system. Calls to specific functions can be redirected elsewhere to functions that are similar (same prototypes) or even to ones that are not (different prototypes). The function mappings that define this capability to redirect from one function to another can be linked together to construct long chains of codes that implement common functionality, but with varying degrees of success for different inputs. With the integration of the DejaVu framework, we have added the ability for temporal adaptivity that allows an application to span the spatial solution domain for problems — reverting the application to a known good state whenever a selected algorithm fails and subsequently learning from these mistakes. These capabilities are illustrated throughout the three scenarios we designed to model simple adaptive policies. We also designed the rudimentary specification language for expressing the adaptive decisions as high-level control decisions. This XML based language allows complex adaptive decisions to be defined quickly and easily relieving the requirement that the user have in-depth knowledge of the system-level constructs.

The second role of our framework is learning the adaptivity that should be applied. For many applications the true, most efficient adaptive policies are unknown because the applications are too large and complex, and the thought that any given adaptive policy may actually decrease the performance of an application can deter anyone from defining policies in the first place. However, using the function interception constructs of this framework, an application can intercept the functioning of a set of codes and randomly explore different paths to determine an adaptive policy over time. By using reinforcement learning algorithms, the application continues to reinforce "smart" decisions while still occasionally exploring others to find the best choices to exploit. In our exper-

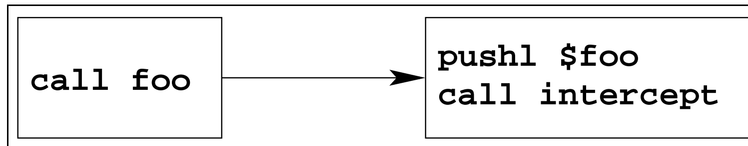

Figure 5.1: Illustration of patching procedure.

imental results we have demonstrated on-line and off-line recommender systems that accumulate knowledge of the best adaptive policies to exploit for optimal efficiency. Therefore, while an adaptive policy may not be known *a priori*, the framework can be used to build one automatically over time.

The following paragraphs demonstrate continued efforts to integrate with additional frameworks to develop powerful new constructs and to finding new applications for our framework.

5.1 Future Work

Work continues on integrating pure binary-level patching into the framework so that we can seamlessly instrument the object files at runtime. Currently, the framework patches the GNU Compiler Collection (GCC) assembler output before it is compiled into the object file. Patching merely requires a simple regular expression matching and replacement script to instrument the files. The exact replacement process is illustrated in Fig. 5.1. However, with this method we still require access to the source code so that we can compile it to assembler, even though we are not making any source-level changes. With further integration with the DynInst [6] framework we plan to support complete, on-the-fly patching so that components can by dynamically linked in and instrumented post-runtime. The DynInst framework handles correcting all the relative jumps and alignments that would be affected by inserting the `pushl` instruction into the executable.

Future work will also look into integrating the framework with another framework, Weaves, the Reconfigurable Programming Framework [34]. Weaves allows arbitrary codes to be composed into a single executing process by relaxing namespace collisions between modules. Typically, multiple executable codes with conflicting variables in their global data contexts are not usable together in a single process. Weaves removes this constraint by separating the Global Offset Tables (GOTs), (tables used for looking up global variables), for each module. This allows our framework to achieve the goal of composing arbitrary scientific codes into a single application without dealing with namespace collisions. While Weaves can separate the GOTs between modules, it can also pick an arbitrary GOT for any executing module. Given this, our framework can support operations similar to *continuation modifiers* — execution can jump to a certain location while maintaining a given data context. This system-level method for supporting continuations is useful for languages that do not naturally include them (e.g., Scheme has the call/cc instruction) to support exception handling and co-routines.

Finally, we will continue to find additional adaptive scenarios from literature and real-world applications, and model them using our framework. We envision this will lead to the formation of new constructs and functionality that can be supported by our framework.

Appendix A

HACL DTD

```
<!ELEMENT hacl (mappings, callbacks)>

<!ELEMENT mappings (mapping*)>
<!ELEMENT mapping (functions, return, parameters)>

<!ELEMENT functions (function+)>
<!ATTLIST function name CDATA #REQUIRED>

<!ATTLIST return value CDATA #REQUIRED>

<!ELEMENT parameters (param*)>
<!ATTLIST param name CDATA #REQUIRED>
<!ATTLIST param type CDATA #REQUIRED>

<!ELEMENT callbacks (callback+)>
<!ELEMENT callback (conditions?,code*)>
<!ATTLIST callback func CDATA #REQUIRED>
<!ATTLIST callback type (pre|post) #REQUIRED>
<!ATTLIST callback mapping CDATA #REQUIRED>

<!ELEMENT conditions (condition+)>
<!ELEMENT condition (remap,rollback?,code*)>
<!ATTLIST condition value CDATA #REQUIRED>

<!ELEMENT remap (assoc*)>
<!ATTLIST remap mapping CDATA #REQUIRED>
<!ATTLIST remap func CDATA #REQUIRED>

<!ATTLIST assoc alloc (exist|global|dynamic) #REQUIRED>
<!ATTLIST assoc from CDATA #REQUIRED>
<!ATTLIST assoc to CDATA #REQUIRED>
<!ATTLIST assoc name CDATA #IMPLIED>
```

```
<!ATTLIST assoc type CDATA #IMPLIED>
<!ATTLIST assoc array CDATA #IMPLIED>

<!ATTLIST rollback search CDATA #REQUIRED>
<!ATTLIST rollback data CDATA #IMPLIED>

<!ELEMENT code (#PCDATA)>
<!ATTLIST code type (pre|POST) #REQUIRED>
```

Bibliography

[1] V. Adve, A. Akinsanmi, J.C. Browne, D. Buaklee, G. Deng, V.V. Lam, T. Morgan, J.R. Rice, G.J. Rodin, P.J. Teller, G.F. Tracy, M.K. Vernon, and S.J. Wright. Model-based control of adaptive applications: An overview. In *Proceedings of Next Generation Systems Program Workshop at the International Parallel and Distributed Processing Symposium (IPDPS'02)*, 2002.

[2] V. Adve, R. Bagrodia, J.C. Browne, E. Deelman, A. Dubeb, E.N. Houstis, J.R. Rice, R. Sakellariou, D. Sundaram-Stukel, P.T. Teller, and M.K. Vernon. POEMS: End-to-end performance design of large parallel adaptive computational systems. *IEEE Transactions on Software Engineering*, 26(11):1027–1048, 2000.

[3] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–159, 1996.

[4] R. Balzer and N. Goldman. Mediating connectors: A non-bypassable process wrapping technology. In *Proceedings of the Workshop on Electronic Commerce and Web-Based Applications at the 19th IEEE International Conference on Distributed Computing Systems*, pages 73–77, 1999.

[5] P.C. Bora. Runtime algorithm selection for grid environments: A component based framework. Master's thesis, Virginia Polytechnic Institute and State University, June 2003.

[6] B. Buck and J.K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[7] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *HPDC*, pages 11–20, 2000.

[8] F. Chang and V. Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing*, 4(1):49–62, March 2001.

[9] T. Elrad, R.E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[10] B. Ensink, J. Stanley, and V. Adve. Program control language: A programming language for adpative distributed applications. *Journal of Parallel and Distributed Computing*, 63(11):1082–1104, November 2003.

[11] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.

[12] T. Gross, P. Steenkiste, and J. Subhlok. Adaptive distributed applications on heterogeneous networks. In *Heterogeneous Computing Workshop*, pages 209–218, 1999.

[13] Object Management Group. The common object request broker: Architecture and specification. http://www.omg.org, 1996.

[14] D. Hiebert. *Manpage of CTAGS*, March 2004. http://ctags.sourceforge.net/ctags.html.

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

[16] B. Jenkins. Algorithm alley. *Dr. Dobbs Journal*, September 1997.

[17] P.J. Keleher, J.K. Hollingsworth, and D. Perkovic. Exposing application alternatives. In *19th IEEE International Conference on Distributed Computing Systems*, pages 384–392, 1999.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[19] S.R. Kohn, G. Kumfert, J.F. Painter, and C.J. Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[20] A.R. Krommer and C.W. Ueberhuber. *Computational Integration*. Society for Industrial and Applied Mathematics, Philadelphia, 1998.

[21] M.G. Lagoudakis and M.L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 511–518, 2000.

[22] K.J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[23] J.P. Loyall, P. Rubel, M. Atighetchi, R. Schantz, and J. Zinky. Emerging patterns in adaptive, distributed real-time, embedded middleware. Technical report, BBN Technologies, 2003.

[24] D.R. Musser. Generic programming. http://www.cs.rpi.edu/ musser/gp/.

[25] D.R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 27(8):983–993, 1997.

[26] NetLib. *QUADPACK Readme*. http://www.netlib.org/quadpack/readme.

[27] N. Ramakrishnan and C. J. Ribbens. Mining and visualizing recommendation spaces for elliptic PDEs with continuous attributes. *ACM Transactions on Mathematical Software*, 26(2):254–273, June 2000.

[28] N. Ramakrishnan, J.R. Rice, and E.N. Houstis. Gauss: an online algorithm selection system for numerical quadrature. *Advances in Engineering Software*, 33(1):27–36, Jan 2002.

[29] R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed. Autopilot: Adaptive control of distributed applications. In *The Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC '98)*, pages 172–179, 1998.

[30] D. Rogerson. *Inside COM*. Microsoft Press, Redmond, WA, 1997.

[31] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

[32] A.A. Stepanov, M. Lee, and D.R. Musser. Hewlett-packard laboratories reference implementation of the Standard Template Library. Source files available from ftp://ftp.cs.rpi.edu/pub/stl.

[33] R.S. Sutton and A.G. Barto. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.

[34] S. Varadarajan, J. Mukherjee, and N. Ramakrishnan. Weaves: A novel direct code execution interface for parallel high performance scientific codes. Technical Report CS.DC/0205004, Computing Research Repository, May 2002.

[35] S. Varadarajan and N. Ramakrishnan. Novel runtime systems support for adaptive compositional modeling in PSEs. *Future Generation Computer Systems (special issue on "Complex PSEs for Grid Computing", to appear)*, 2004.

[36] J. Veitch and R. Laddaga. Distributed dynamic systems - introduction. *Communications of the ACM*, 41(5):34–36, May 1998.

[37] M. Yarvis, P.L. Reiher, and G.J. Popek. Conductor: A framework for distributed adaptation. In *Workshop on Hot Topics in Operating Systems*, pages 44–, 1999.