DITools: Application–level Support for Dynamic Extension and Flexible Composition*

Albert Serra, Nacho Navarro, Toni Cortes
Departament d'Arquitectura de Computadors
Universitat Politecnica de Catalunya
{alberts,nacho,toni}@ac.upc.es

Abstract

Today, operating systems set–up process images from executable files using fixed rules. Programs are restricted to run in essentially the same environment at every execution. However, we believe that this behavior is not always convenient, and that many times it is interesting to make variations to the execution environment, either to introduce new functionality or to specialize critical services, even when their source code is not available. This problem can be mitigated through application–level extensibility and flexible composition of binary modules.

In this paper, we describe DITools an application–level tool that supports dynamic interposition on dynamically–linked procedure–call boundaries. This tool enables both global and per–module dynamic interposition. We also present a detailed use of DITools and various short examples of extensions.

1 Introduction

Modifying programs, libraries and operating systems becomes a difficult problem when the source code is not available, and this is a common situation. Vendors are quite reticent to give away the source code of their applications and/or operating systems. On the other hand, there are many situations in which these modifications would be very useful. Let us examine some examples.

Tracing and monitoring applications is a common technique used to learn the behavior of applications. To perform this task, some code has to be added to the program to trace or monitor the desired events. If the source code is not available, there is no easy way to do it.

Another problem arises when trying to enhance proprietary applications to take advantage of parallel resources such as multiple CPUs. Let us suppose an application that invokes many time–consuming operations from a sequential mathematical library, for example large FFTs. If we do not have the source code, changing the implementation of the FFT to make it a parallel one, would probably mean to write a whole new library. If we only want to enhance the FFT, it is going to be very difficult without the source code of the application or the library.

If we focus our attention on the operating system execution environment, it is very difficult to modify its functionality (for instance, to build a new file system) without OS support. Even if there is some OS support, we will probably need the help of the system administrator to install our enhancement. There is no easy way to do a rapid prototyping that is being used by many applications without the help of the system administrator.

Finally, sometimes we would like to modify an application to send and receive the data in an encrypted form. If we do not have the source code of the application or library, this modification will be a big problem.

In this paper, we present a tool that can be used to solve all the above problems. This tool provides transparent user–level extensibility and flex-

*This work has been supported by the Ministry of Education of Spain (CICYT) under contract TIC98-0511, and by the Comissionat per a Universitats i Recerca de la Generalitat de Catalunya under the grant FI96-3088

ible composition within applications. This allows any user to load new code and to interpose it between the call and the definition of functions, making possible to modify or extend programs without rebuilding them. We also describe several examples in which we use the tool to solve the aforementioned problems.

This paper is structured as follows. Section 2 gives an overview of our approach to dynamic extension and flexible composition, describing the framework and the tool. Section 3 discusses performance issues and presents an evaluation. Section 4 contains the examples. Section 5 reviews related work and, finally, Section 6 summarizes and concludes the paper.

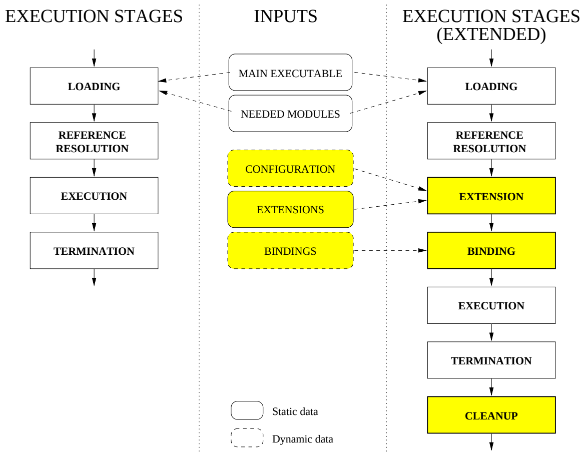

Figure 1: Usual and extended execution stages

2 Dynamic Extension and Flexible Composition

In order to build the process image for a given executable file, modern operating systems need to glue together multiple modules (dynamically–linked libraries) at program load time. This happens because programs are not self–contained in terms of functionality.

The system provides service definitions to complete the image. These definitions are located using fixed resolution policies. This results in essentially the same execution environment at every run of the same program. However, as we illustrate in the introduction, there are situations in which it can be very helpful to make variations in this image. Moreover, these variations should not be limited by source code availability, nor by arbitrary rules embedded in operating system components. This motivates our interest on making the process of building the runnable image more controllable by users and programs themselves.

The goal of our research is to enable 'ad–hoc' execution environments, by allowing applications to build appropriate execution environments for themselves, according to performance requirements, efficiency concerns and functionality needs. This covers the improvement or tuning of services (e.g. service specialization or result 'memoization' – see [17]), the addition of new functionality to do data stream processing (e.g. encryption or compression) or to perform new tasks (e.g. cooperation with resource managers), and even restructuring the execution environment (e.g. code co–location and distribution).

In Figure 1 we show the phases in which we conceptually divide the process of executing a program. Two scenarios are shown: the usual one (left side) and the extended one (right side). The meaning of each stage is as follows: *loading* brings all the required modules into the address space, *reference resolution* finds definitions for unresolved references, *extension* introduces new modules in the address space, *binding* allows to customize bindings, *execution* enters application code, *termination* exits application code and *cleanup* shuts down extensions. For each stage, we show which inputs determine its behavior. The stages shaded and surrounded by thick lines are provided by our tools.

In the *extension* stage, we check for the need of extending the image using configuration information that can change at every run. If extensions are needed, the right modules are introduced in the process image. Then, the *binding* stage arranges these modules to be executed by setting up bindings appropriately. This stage is independent from the previous one. Therefore, it is possible to modify bindings without loading extensions.

The tools described in this paper introduce an extension to the runtime loader and linker. This extension provides basic functionality to load and bind unforeseen modules and services, as well as to reconsider actual bindings. This functionality works entirely at user–level and complements the traditional dynamic loading and linking services. This is achieved by means of additional stages after program loading, devoted to adapt/extend the execution environment. These tools also export services, making them available during execution, to allow on–the–fly adaptation.

These tools are being used by various projects in our department for trace collection, scheduling research and I/O research. They have helped us to test the stability and validity of our services and abstractions.

2.1 Environment

Modern operating systems promote the use of shared libraries for efficiency. Linking against shared libraries results in smaller program sizes on disk, and also requires less physical memory at run time. Moreover, shared libraries can be fixed and improved without rebuilding executables.

The use of shared libraries leads to executable files containing unresolved references. At program loading time, the system should load the libraries upon which executable files depend, and then it should fix unresolved references to point to the right definitions. This is called *dynamic linking*.

Dynamic linking requires cooperation between development tools, object file formats and operating system components. For example, a common technique for supporting dynamic linking is by making indirections through *linkage tables*, which are placed in the executable file by the static linker, and then used by the dynamic linker at load time. Usually, the system's Application Binary Interface (ABI) defines the object file format and these data structures.

During the last years, many vendors are adopting the System V r4 ABI [4], or at least parts of it. IRIX, Solaris, Linux, FreeBSD or HP-UX currently support ELF as the object file format and use similar dynamic linking conventions. ELF is the file format originally introduced in System V r4.

The standardization of binary interfaces makes possible to take a generic approach for extending the available functionality (we currently have a running version for IRIX and Linux sharing a 70% of source code), as well as for manipulating bindings both at program loading time or at run time.

2.2 Exploiting dynamic loading

Shared libraries make dynamic loading of code easier. In fact, many operating systems provide interfaces to load this kind of binary modules at run time. We exploit these services to introduce extension code in the process image (although we improve some aspects of conventional dynamic loading, as explained in following sections).

The mechanisms used to gain control at program startup can vary from system to system, but they are reasonably uniform and widespread in the UNIX world. In most cases, this is as simple as setting an environment variable (`LD_PRELOAD` in Linux and Solaris or `_RLD_LIST` in IRIX, for example) pointing to the module to be loaded, and to declare an 'init routine' in this module. At program startup, this module will be loaded within the process image, and the init function will be executed before the program itself. For security reasons, the system loader disables this feature when loading setuid programs.

2.3 Dynamic interposition

Conceptually, 'interposition' is the addition of functionality at the midst of an existing interface boundary. This mechanism is appropriate for binding extensions, because it exploits existing interface boundaries to attach new services while preserving old ones.

We deal with interposition at procedure–call boundaries. That is, we add functionality in the program execution path, between references to procedures and the procedures themselves.

To achieve this kind of interposition dynamically, at run time, we need to detect references to procedures and to be able to change their target definitions within the process image. Dynamic linking draws a clean boundary between external references and their definitions, by means of the linkage tables and other data structures described above. In this work we exploit these data structures for interposition purposes.

2.4 DITools

The infrastructure required to manage dynamic loading of extensions and interposition is provided by DITools (our Dynamic Interposition Tools). DITools allows modifying the execution environment of dynamically–linked executables at every run, either to select among different service implementations, as well as to accommodate unforeseen functionality. Using this infrastructure, programmers are allowed to change bindings, redefine symbols, load new modules or remap global variables.

Architecture

DITools is structured around a runtime module (DI runtime) that cooperates with the dynamic loader and linker to support extension and flexible composition. Once the program, together with all its dependencies, has been loaded by the system, the DI runtime gains control and performs some post–load processing (see figure 1), according to the needs for this individual run. This is done using existing mechanisms, as mentioned in section 2.2.

It is interesting to note that this tool works entirely at user level, without kernel support, and without administrator privileges. The scope of the tool can be easily controlled, so it can manage from single programs to entire sessions and multiple users.

DITools processing can be driven by a configuration file or it can be driven using a programming interface, or some combination of both approaches. DITools allows two main classes of operations: loading of new functionality and manipulation of bindings. Both kinds of operations will be explained in detail in the following sections.

Loading of new functionality

During the extension stage, as well as at run time, DITools can load arbitrary extension modules (also called backends in our framework) within the process image. These modules can be declared in a configuration file, to keep the executable file isolated from this kind of 'volatile' decisions.

We have adopted the shared–library model for these extensions, which means that our extensions are shared libraries themselves. This has two basic advantages. The first one is that we can use the existing development tools. And second, as our extensions are also shared libraries, we can use DITools recursively to extend them.

The support provided by DITools allows modules to be loaded multiple times within the same process (i.e. replicated) without symbol collisions.

Rebinding and redefinition

We exploit the dynamic–linking data structures for interposition purposes. DITools supports explicit manipulation of bindings between dynamically–linked references and definitions exported by modules. It currently implements two mechanisms: the first one allows changing the target for a given reference (we call it *rebinding*), and the second one allows 'wrapping' (i.e. mediate all the uses of) a given definition (we call it *redefinition*). While the effects of the first mechanism are selective and leave the original definition untouched, the second one affects all the uses of a definition, effectively hiding it to the outside. Using rebinding we can achieve *selective overriding* of definitions, while using redefinition we are doing *global overriding* of definitions.

References and definitions are identified by a pair (module, name). So, a single rebinding can be specified as follows:

```
(program,read) -> (mymodule,myread)
```

Meaning that references to 'read' done by the module 'program' should point to the definition 'myread' in the module 'mymodule'.

As can be observed, this mechanism is independent of conventional resolution policies based on name matching. This makes possible the coexistence of definitions with the same name in different loaded modules.

Acting over references and definitions individually also proves to have its advantages. On the one hand, we can avoid affecting all the uses of a definition when inserting new functionality. On the other hand, we do not require to subclass the entire library in order to interpose on its interface.

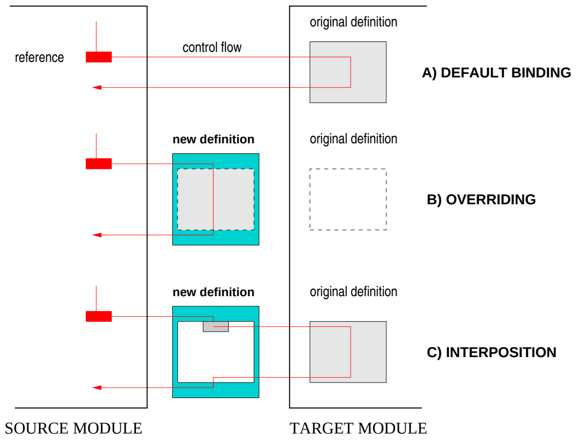

Figure 2: Scenarios after rebinding references

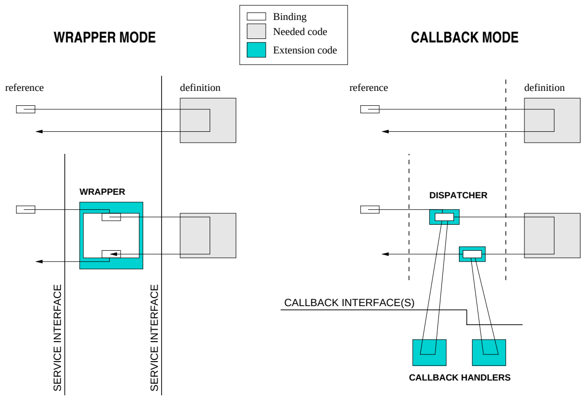

Figure 3: Wrapper and callback modes of extension execution

Interposition on top of DITools

Figure 2 shows some possible scenarios after rebinding. The upper one (A) depicts how bindings are by default, in which pointers in the reference side make possible to reach definitions residing in another module. The central one (B) shows how changing bindings in the reference side can be used to override definitions in a per–module basis. The lower one (C) shows how to achieve interposition, by changing the binding while providing another one to invoke the previous definition.

Therefore, DITools can be used for interposition of extensions. This requires changing a binding on the reference side to point to the extension, and another one within the extension code to point to the original definition. For example:

```
(program,read) -> (mymodule,myread)
(mymodule,read) -> (libc,read)
```

The above lines add 'mymodule.myread' as the target for references to 'read' being done by the module 'program'. References to 'read' coming from 'mymodule' are conveyed to 'libc.read'. Although this last rebinding is unnecessary in the case in which it refers to the conventional definition, we show it for clarity.

Extension execution modes

To simplify design and implementation of extensions, DITools allows two modes of extension execution: *wrapper* and *callback*. The *wrapper* mode can be seen as 'plain interposition'. This mode requires providing routines that match the interface on which they will be interposed. The *callback* mode is a refinement that allows a routine to be invoked regardless of the interface being intercepted. This refinement is especially convenient when building extensions for monitoring purposes, as well as when the interface is not known.

Both modes are depicted in Figure 3. The *wrapper* mode is shown in the left side of this figure. Observe that the wrapper is invoked by the application as the effective service definition. Therefore, the wrapper should present the same interface to the application. The *callback* mode appears in the right side of the figure. In this mode, DITools transparently interposes a *callback dispatcher* instead of the extension routine. This dispatcher will invoke the routines 'aside' of the service definition, without need of knowing the implied interface. Callback handlers have a fixed interface (as illustrated in Section 4), which is determined by the callback dispatcher that calls them. The dispatcher is also the responsible of invoking the original service definition, as if nothing happened.

Extensions can provide their own dispatchers to enable different ways of invoking extension code, for instance to allow multiple handlers for the same event, as in other frameworks [14] [15].

Dynamic adaptation facilities

Once the application has been started, it is still possible to make changes in the execution environment, given that both loading and bind manipulation facilities are available through the interface of the DI runtime. By means of these dynamic facilities, we can design lightweight extensions to watch for some conditions before activating new functionality. At run time, these extensions can also deactivate the new service dynamically.

Let us suppose that we have installed an optimized read path for files with read–only access. The following code illustrates how to deactivate the optimization when conditions change. The wrapper has been installed to be activated in response to references to 'open()'. It checks for the O_RDONLY flag, and deinstalls the new paths when a file is opened with flags that can compromise the extension:

```
int rdopen_wrapper(char *file, int flags)
{
  if (flags&O_RDONLY)
    di_rebind("program","read",
              "extension","rdo_read");
  else
    di_rebind("program","read",
              "extension","rdwr_read");

  return(open(file, flags));
}
```

The DITools interface provides rebinding, symbol redefinition, interrogation about current bindings and loaded modules, among other operations. This interface is described in more detail in a research report [18].

Process management facilities

Startup: When extension modules are loaded, DITools allows them to initialize before being invoked by the process. This can be used to allocate data structures or to set–up some bindings dynamically.

Cleanup: At the end of execution (for instance, when the process exits or execs another binary), DITools gives a chance for doing cleanup work within extension modules (e.g. write data structures to disk, close any private handle or free other allocated resources). During extension cleanup, the DI runtime shuts down rebindings and redefinitions in order to guarantee that extensions do not indirectly trigger themselves.

Image changes: Actions that cause image changes (e.g. dynamic–loading requests) are captured by the DI runtime in order to preserve the behavior expected by extension modules. For example, DITools checks that new modules do not affect existing rebindings.

Fork: A default behavior that can be deactivated if required is that forked processes inherit the DI runtime, as well as rebindings and redefinitions existing in the forking process.

2.5 Considerations

The support described above has some natural limitations. First of all, the mechanism cannot be used in statically–linked binaries. On the other hand, the set of available dynamically–linked procedure calls may not be enough for some uses that require customizing pieces smaller than functions. Also, some bindings may not be available for manipulation due to static optimizations. In these cases, the scope of the tools can be complemented using binary rewriting techniques (e.g. Paradyn's dynamic instrumentation [22]) to invoke the backend for events other than dynamically–linked procedure boundaries. However, some systems also use linkage tables to support Position Independent Code. This makes possible to use DITools to control intra–module bindings in addition to external references.

Finally, there are also not–so–obvious considerations to take into account when using all the facilities provided by DITools. The design of existing library services may be inappropriate for dynamic rebinding, because they may keep state between invocations. On the other hand, there are situations in which modules can shortcut linkage tables used in cross–module invocations (e.g. by assigning an address to a pointer and then invoking it directly).

3 Performance

The DITools infrastructure enables qualitative benefits, namely richer functionality and ease of extension, that are hard to measure. In this section, we will focus on the overhead of adding new functionality, regardless of the beneficial counterparts, and we found it quite reasonable. Nevertheless, we believe that performance is not always the primary concern.

We have designed a worst–case experiment (the 'Forward' experiment) that redefines all the available dynamically–linked definitions within a given process to a simple extension that merely forwards the call to the original definition. This includes all definitions of the standard C run–time library as well as those included in any other library used by the program. By comparing the results to the execution of the unmodified program (the 'Baseline' experiment), we evaluate the overhead of using DITools. Typical uses of DITools (like those described in section 4) only need to interpose code to very few calls.

The programs used in our tests come from the SPEC95 benchmark suite, using the 'train' input dataset. Our experimental environment is based on a 64–processor Origin2000 system, from Silicon Graphics Inc. This machine runs the IRIX Operating System, release 6.5.3.

In programs enhanced by our infrastructure we expect to observe two effects on performance: an increase in startup time due to extra processing, and some overhead during execution due to the additional indirection.

Results summarized in Table 1 come from the average of 4 executions of each benchmark, running on a dedicated processor of the machine, to minimize the effects of cold start and interferences from other processes. This table shows, for each program, the number of statically available dynamically–linked references (*static hooks*), how many times these hooks are triggered at runtime (*dynamic activation count*), the elapsed time for the unmodified execution environment (*baseline*), and the elapsed time when invoking the empty wrapper at every call.

The *static hooks count* column gives an idea of the work done by the DI runtime at startup. As many bindings as listed in this column are modified to invoke the extension. This classifies the benchmarks in three groups, according to the number of exposed hooks. Benchmarks that need the same libraries expose the same number of hooks. Given that the *startup* time is basically constant (around 30 mi-

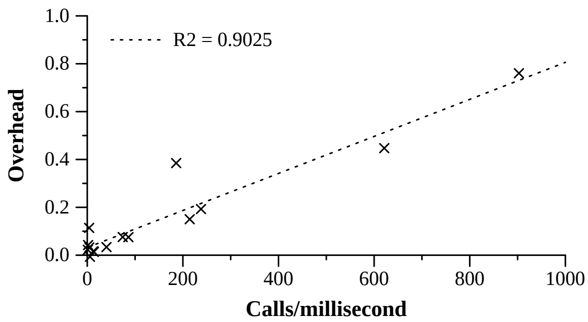

Figure 4: Overhead vs. calls per millisecond

croseconds/hook), we do not show it in the table. It ranges from 140 ms for the first five benchmarks (3,500 exposed hooks) to 150 ms for the last 8 benchmarks (5,600 exposed hooks).

Last three columns show the elapsed time for *baseline* and *forward*, and the corresponding *overhead*. The overhead of DITools is proportional to the dynamic count of hook calls. In many cases it is less than 10%. However, benchmarks that are making a high number of calls to the extension, relative to their execution time, become more affected by interposition. This is simply the scaled effect of this high number of calls per millisecond (c/ms). For instance, if we compute this number for perl and su2cor, we obtain 902 c/ms for perl and 621 c/ms for su2cor. Both programs follow the correlation between calls per millisecond and overhead seen in the other programs. Take, for instance, turb3d, which has an overhead of 3%. It does 40 c/ms, so this makes 1,013 c/ms for 76% and 613 for 46%, which are close to the values observed for perl and su2cor. The measured correlation coefficient between overhead and calls per millisecond is 0.9 (see the regression plot in Figure 4). It is worth to mention that we are describing a worst–case experiment, in which the extension interposes to all the available hooks.

Table 2 depicts the space requirements of the DI runtime and the other extensions used in this evaluation, as well as the average size of the benchmarks. This table shows, for each module, its static size in disk as well as the size of all the virtual memory regions required to hold its code and data within the process address space at run–time. The 'count' extension has been used to compute the columns labeled 'hook counts' in Table 1, the 'forward' extension corresponds to the 'forward' experiment, and the 'monitoring' extension to the 'monitoring' experiment.

	Hook counts (/1,000)		Elapsed time (ms)		Overhead
Benchmark	Static	Dynamic	Baseline	Forward	%
Go	3.5	5.8	5,002.3	5,115.0	2%
M88ksim	3.5	4.8	864.6	858.8	0%
Gcc	3.5	289.3	1,555.9	2,154.7	38%
Compress	3.5	0.7	181.5	202.2	11%
Ijpeg	3.5	12.3	2,385.6	2,486.7	4%
Li	3.9	4.3	975.8	991.6	2%
Perl	3.9	12,335.0	13,662.2	24,051.4	75%
Tomcatv	5.6	3,412.1	39,599.9	42,582.3	8%
Swim	5.6	5,284.1	2,466.2	2,836.9	15%
Su2cor	5.6	24,504.6	39,447.5	57,090.1	45%
Mgrid	5.6	319.0	23,107.2	23,425.7	1%
Applu	5.6	4.3	1,020.8	1,053.8	3%
Turb3d	5.6	1,600.7	39,811.9	41,159.9	3%
Fpppp	5.6	50.1	676.9	728.2	8%
Wave5	5.6	2,002.2	8,417.1	10,039.7	19%

Table 1: Impact of interposition on program execution

Module	Static size	Dynamic size
DI runtime	64K	1.5M
Count extension	29K	0.5M
Forward extension	30K	0.5M
Monitoring extension	30K	1M
SPEC average	430K	10M

Table 2: Space requirements of modules used in this evaluation

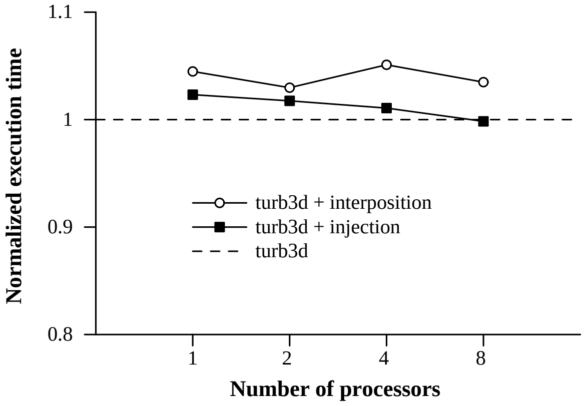

Figure 5: Impact of a fully–featured extension on execution time

The dynamic sizes have been obtained by measuring the number of pages allocated in the virtual process address–space, and then using the page size (16K) to compute the dynamic size. The space overhead at run–time is about 15%. We should take into account that benchmarks are running with a relatively small input ('train').

The second experiment (the 'Monitoring' experiment) compares the performance of a fully–featured extension using DITools, against the same functionality introduced by changing the source code. The experiment introduces a performance monitoring extension that collects the execution trace of a parallel program. This trace contains thread creation, thread joining and synchronization events. More information on this experience can be found in another paper [10].

For this experiment, we use a parallelized version of the turb3d benchmark. Figure 5 shows the impact of both alternatives. The dashed line represents the normalized execution time for the uninstrumented version of this benchmark, using from 1 to 8 processors. Solid lines represent the normalized execution time for the instrumented versions. As can be observed, interposition–based instrumentation performs comparably to the static code modification approach. In both cases, the overhead is less than 5% of the execution time for any number of processors.

At a first glance, one can think that all these enhancements always come at the price of performance. In another experiment, we used interposition to enhance the performance by caching the results of frequently used functions [17]. In this way we were able to reduce the execution time by 8% for those library functions. This demonstrates that extending the execution environment not always degrades performance.

Finally, remember that, although this section focuses on the costs of adding new functionality using DITools, this infrastructure is intended to support extension and flexible composition. Therefore, performance will not always be the primary concern. In many cases, the infrastructure will not be used to *add* functionality but to *select* among multiple implementations (thus, not adding extra indirections), and it may even pay using it in terms of performance.

4 Examples

In this section, we discuss how to build a couple of useful modules using DITools to facilitate a better understanding of the framework. We begin with a full example, and then we give short ideas about how to use DITools to build some additional extensions.

4.1 Program monitoring

Interposition can be used to understand the behavior of programs by capturing information of service invocations. In this example, we describe an extension used to monitor interactions between a program and its runtime. This extension will provide an execution profile containing function entry and exit information, and the corresponding timestamps. Once processed, the output of the extension can give a profile like the following one, coming from the execution of the NAS BT benchmark on a SGI Origin 2000:

```
...
2623262   +178 x_solve_
2623486     +175 lhsx_
2638509     -175
2638738     +186 x_solve_cell_
2673173     -186
2673412     +185 x_backsubstitute_
2674186     -185
2674361   -178
2674493   +191 y_solve_
```

```
2674630      +176 lhsy_
2678248      -176
2678449      +199 y_solve_cell_
2741176      -199
2741403      +198 y_backsubstitute_
2741949      -198
2742171    -191
...
```

The first column in this profile contains the event timestamps in nanoseconds, followed by a sign (+/-) indicating procedure entry/exit, then the event identifier, and finally the event name (in this case, a procedure name). Tabulation represents invocation nesting.

In this extension, we decided to use the callback mode (see 'Extension execution modes' in section 2.4), because it allows to capture all the references using only two handlers.

The steps to be followed to build and install this extension can be summarized as follows:

1-Write the event selector function. When using the DITools callback dispatcher, the user should provide a routine (`di_callback_required`) in the backend to select which hooks will trigger the user–provided callback handlers, and to give an identifier for each possible event. The function will be invoked at extension time (see section 2 for a description of the stages) once for each potential event. A return value of zero can be used to ignore the event. The event identifiers will be passed at execution time to the callback handler.

```
int di_callback_required(char *func)
{
  static int event_id=0;

  event_id++;
  funcs[event_id]=RECORD_FUNC_NAME(func);
  return event_id;
}
```

2-Implement callback handlers and support routines, i.e. the code which should record the time and the event. Callback handlers will be invoked by the callback dispatcher before (`di_pre_handler`) and after (`di_post_handler`) dynamically–linked hooks selected by the event selector function. Callback handlers receive two arguments from the dispatcher: a virtual processor identifier and an event identifier. In this example, we use a macro (PUT_EVENT) to record time–stamped events in a buffer managed by the extension code. This buffer is set–up at startup time (`di_init_backend`), and processed at the end of execution (`di_fini_backend`).

```
event_t *buffer;
char *funcs[MAX];

int di_init_backend()
{
  buffer=ALLOCATE_BUFFERS();
  return buffer!=NULL;
}

void di_pre_handler(long vpid, long event_id)
{
  PUT_EVENT(vpid, event_id, START);
}

void di_post_handler(long vpid, long event_id)
{
  PUT_EVENT(vpid, event_id, END);
}

void di_fini_backend()
{
  int fd=open(tracefile, flags);
  PROCESS_TRACE(fd, funcs, buffer);
  close(fd);
}
```

Note that backend portability is determined only by the extension code, since there are no platform–specific details in the DITools interface. Platform–dependent pieces (e.g. the callback dispatcher) are provided by the DITools runtime.

3-Build the extension module, by compiling the above code like a standard shared library.

```
cc -shared -o progmon.so progmon.c
```

4-Write the configuration file to be parsed by DITools at extension time. It should declare which backend should be loaded (1) and specify that our backend routines should be invoked as *callbacks* at every reference to dynamically–linked runtime services (2).

```
// begin of backends section
// (1) request the module "progmon.so":
BACKEND backends/DIFlow/progmon.so
// end of backends section
#commands
// begin of commands section
// (2) request the installation of
// the callback dispatcher at every
// dynamically-linked reference done
// by the MAIN module
MAIN *      DIRUNTIME callback_dispatcher
// end of commands section
```

5-Run your unmodified program. Specify your config file and set–up the system to load the DI run-time at program startup. This last step is system dependent, as explained in 2.2.

```
$ setenv DI_CONFIG_FILE progmon.conf
$ setenv LD_PRELOAD diruntime.so
$ <your program>
```

4.2 Service improvement

In this example, we replace the definition of FFT used by a program, by another one that makes a more efficient computation exploiting multiple CPUs. Given the new FFT service, the extension should simply bridge differences in the interface:

```
void fft_bridge(float *data, long size)
{
  n = get_num_processors();
  new_data = reshape_data(n, data);
  spawn(n, parallel_fft, new_data, size);
  wait_for_end();
}
```

This routine computes the available number of processors, prepares the data to be used by the threads, and spawns threads to execute a parallel FFT using the reshaped data. Once built, we should simply override the old FFT service using the redefinition facility, either through the config file or at run–time (`di_rebind`).

4.3 Filesystem extension

The third example is a filesystem extension that allows applications to transparently access remote files. Requests corresponding to remote filesystems are redirected to a server running elsewhere. The extension should provide wrappers for filesystem services, analyze the arguments and then choose which underlying service should be invoked to complete the request.

Usually, dynamically–linked programs do not trap directly to the operating system for reading and writing. Traps are usually encapsulated in a system library (e.g. libc) and are exported as library functions. Therefore, DITools can interpose the above routines to these functions to achieve the expected behavior, without the need of system–call redirection functionality.

```
int fs_write(int fd, char *b, int s)
{
  if remote_fd(fd) {
    a = marshal(b,s);
    r = send_request(server, WRITE, a);
  } else
    r = base_fs_write(fd,b,s);
  return r;
}
```

4.4 Data stream processing

In this last example, we illustrate how to extend I/O services to encrypt and decrypt the data stream. Given the appropriate encryption algorithm, the extension should simply provide the routines that combine in the right order the encryption/decryption process and the I/O operation:

```
int crypt_read(int fd, char *b, int len)
{
  int r;
  r = read(fd, b, len);
  if (encrypted_channel(fd))
    decrypt(b, r);
  return(r);
}

int crypt_write(int fd, char *b, int len)
{
  int r;
  if (encrypted_channel(fd))
    crypt(b, len);
  r = write(fd, b, len);
  return(r);
}
```

These routines can be used to redefine previous read/write services. To make encryption transparent to the program, channels to be encrypted can be determined by the extension. For example, it can decide to encrypt only those data streams that are sent or received from the network.

5 Related work

In software environments that are built from separate modules, interactions frequently happen to be limited to well–defined interfaces between modules. Being able to interpose functionality in the midst of these boundaries, preserving physical encapsulation, has been recognized as very convenient from the extensibility point of view. This has motivated many different approaches to the problem, which are reviewed in this section.

In the first place, many systems provide interposition facilities for the system–call interface. Classical microkernels like Mach[1] or recent kernels like Pebble[7] are examples of it. It is possible to use these facilities to enable profiling (like strace in Linux) as well as functionality extension (like UFO[2]). The usefulness of this kind of interposition has been the motivation for developing extension–enabling frameworks using system–call redirection, like COLA[14], Interposition Agents[13] or SLIC[9].

Our approach differs from the above extension–enabling frameworks in that we do not rely on operating–system provided interposition services. Moreover, we are providing these facilities for interfaces between user–level modules.

Many of the techniques used inside operating–system kernels in order to dynamically accommodate functionality (like device drivers, stackable file systems [11] or other kernel modules) are working examples of extensibility and flexible composition. However, these techniques are focused to specific kernel interfaces, and its use is typically restricted to privileged users. There is a considerable amount of research on building *extensible systems*, a kind of systems that would allow generic, safe, and application–specific extensions (e.g. SPIN[5], VINO[16]), although this research is biased towards safety concerns of in–kernel extensions.

In contrast to the above techniques, we are taking a generic approach. Our goal is to enable extensions to be attached at any available interface, by providing convenient binding mechanisms for these extensions. In addition, we are looking at backwards–compatible ways of extending the execution environment of today programs, without new kernel services.

Finally, there are also approaches to structured dynamic extension at user–level procedure–call boundaries. We can found it within advanced runtime environments (also known as "component platforms"), like ORB implementations [8] and COM+ [21]. These platforms allow subclassing of binary components, and their mechanisms (cross–component inheritance, containment or aggregation) can be seen as specialized forms of interposition. These platforms aim to provide component interoperability.

DITools derives its extension and dynamic loading abilities directly from the system's ABI, exploiting physical encapsulation, and being largely independent of language and program development issues. In contrast, component platforms are tied to language encapsulation, and they usually need to enforce boundaries using object–oriented programming and interface definition languages. They are neither appropriate to extend nor to customize the execution environment of each and every program that runs in a given system.

We found also tools oriented to enable interposition aside of any object middleware, like Detours for Win32 [12] and SLI for Solaris [6]. In Detours, function calls are dynamically intercepted by re–writing function images in order to redirect the control flow to different locations. In contrast, SLI interposition is based on symbol preemption in the resolution mechanism. In this way, SLI can dynamically introduce profiling and tracing functionality into dynamically–linked programs without changing the program image. Last releases of Solaris include support for modifying bindings done by the runtime linker [19].

Tools like Detours and SLI are similar to DITools in that they work at application–level and provide similar interposition facilities. However, they exhibit limitations on their abilities to control definitions and bindings that do not exist in our framework. In particular, we allow selective rebinding of references, while these tools can only do global redefinitions. Detours requires altering the text pages and performs very intrusive changes in the program (e.g. it changes memory references for the original definition). Symbols redefined by SLI will persist until the program exits. On the other hand, the Solaris runtime–linker support is a platform–specific approach that focuses on process monitoring.

6 Conclusions

In this article, we describe DITools, an infrastructure devoted to extend applications at run time, and to tune execution environments by dynamically changing bindings between binary modules.

DITools works at application level, and does not require superuser intervention. Moreover, it uses standard object file formats and common tools, and preserves backwards compatibility. We believe that DITools demonstrates to which point current loading and execution services can be improved to ease the job of researchers, developers and users.

We have also evaluated the overhead of DITools. The evaluation shows that it depends on the dynamic usage of extensions. We believe that this overhead can be tolerated in the most common uses of this tool.

DITools is currently being used for monitoring and trace collection purposes, as well as to build research prototypes for scheduling and I/O research. A distribution of the tools is available for research and academic purposes. Please consult `http://www.ac.upc.es/recerca/CAP/DITools`.

7 Acknowledgments

The authors would like to thank the Nanos project, the CEPBA-UPC team, and the I/O group, for performing the testing of the tools, as well as for their valuable feedback. Specially Xavier Martorell with its CPU manager, and Jordi Caubet with MPTrace. We also thank the GSOMK people for encouraging this research, and the CEPBA-UPC staff for its help during the evaluation.

Finally, we would also like to thank the anonymous referees for their insightful comments, as well as the USENIX shepherd Liuba Shrira for its help during the revision of the paper.

References

[1] M. Accetta, R.Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young "Mach: A New Kernel Foundation for UNIX Development", In Proc. of the Summer USENIX Conf., Aug. 1986

[2] A. Alexandrov, M. Ibel, K. Schauser, C. Scheiman "Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System", ACM TOCS, Aug. 1998

[3] J. Arnold "ELF: An Object File to Mitigate Mischievous Misoneism", In Proc. of the Summer USENIX Conference, 1990

[4] AT&T "System V Application Binary Interface", Published by Prentice–Hall, 1990

[5] B. Bershad et al. "Extensibility, Safety and Performance in the SPIN Operating System", In Proc. of the 15th SOSP, 1995

[6] T. Curry "Profiling and Tracing Dynamic Library Usage Via Interposition", In Proc. of the USENIX Summer Technical Conference, 1994

[7] E. Gabber, C. Small, J. Bruno, J. Brustoloni, A. Silberschatz "The Pebble Component–Based Operating System", In Proc. of the USENIX Annual Technical Conference, June 1999

[8] O. Gampel, A. Gregor, S. B. Hassen, D. Johnson, N. Jnsson, D. Racioppo, H. Stllinger, K. Washida, L. Widengren "IBM Component Broker Connector Overview", Document Number SG24-2022-02, IBM Corp. 1998

[9] D. Ghormley, S. Rodrigues, D. Petrou, T. Anderson "SLIC: An Extensibility System for Commodity Operating Systems", In Proc. of the USENIX Annual Technical Conference, June 1998

[10] M. Gonzalez, A. Serra, X. Martorell, J. Oliver, E. Ayguade, J. Labarta, N. Navarro "Applying Interposition Techniques for Performance Analysis of OpenMP Applications", In Proc. of IPDPS'00, May 2000

[11] J. Heidemann, Popek "File System Developmnent with Stackable Layers", In Proc. of ACM TOCS, Feb. 1994

[12] G. Hunt, D. Brubacher "Detours: Binary Interception of Win32 Functions", Microsoft Research, Technical Report MSR-TR-98-33, Feb. 1999

[13] M. Jones "Interposition Agents: Transparently Interposing User Code at the System Interface", In Proc. of the 14th ACM Symposium on Operating System Principles, Dec. 1993

[14] E. Krell, B. Krishnamurthy "COLA: Customized Overlaying", In Proc. of the Winter USENIX Conference, 1992

[15] P. Pardyak, B. Bershad "Dynamic Binding for an Extensible System", 2nd OSDI Symposium, Oct. 1996

[16] M. Seltzer, Y. Endo, C. Small, K. Smith "Dealing with disaster: Surviving misbehaved kernel abstractions", 2nd OSDI Symposium, Oct. 1996

[17] A. Serra, X. Martorell, N. Navarro "Dynamically–linking Extensions and the Memoization Experience", Technical Report UPC-DAC-1999-17, 1999

[18] A. Serra, N. Navarro "Extending the Execution Environment with DITools", Technical Report UPC-DAC-1999-26, 1999

[19] Sun Microsystems Inc. "Linker and Libraries Guide", Solaris 2.6 Software Developer Collection, 1997

[20] D. Orr, R. Mecklenburg "OMOS - An Object Server for Program Execution", In Proc. of the IWOOOS, 1992

[21] D. Platt "Understanding COM+", Microsoft Press, 1999

[22] Z. Xu, B. Miller, O. Naim "Dynamic Instrumentation of Threaded Applications", In Proc. of 7th ACM SIGPLAN, May 1999