An Extender's Guide to the Optimization Programming Interface and Target Descriptions

Release version 2.02.07.15

Glenn Holloway and Michael D. Smith
{holloway,smith}@eecs.harvard.edu
Division of Engineering and Applied Sciences
Harvard University

July 15, 2002

Abstract

The optimization programming interface (OPI) is a programming interface for use in writing portable analyses and optimizations that work on code represented at or near the machine level. This document describes the structure of the OPI with respect to the libraries that implement it, and it presents several examples of how you can extend the OPI by adding new data structures and/or functions. It also discusses the implementation and extension of the target libraries.

Contents

1 Introduction

The optimization programming interface (OPI) is a set of standardized data structures (that represent the code being optimized and the characteristics of the compilation target) and functions (that create, inspect, and manipulate those data structures). It allows you to express machine-dependent optimization algorithms in a readable and reusable form. By eliminating specific references to the underlying compilation system (called the *substrate*) and current compilation target, optimizations based on the OPI are easy to re-target and port. This document describes the structure of the OPI with respect to the libraries that implement it, and it serves as a reference manual for those interested in extending the OPI.

The OPI is not a complete compilation infrastructure. It is designed to be implemented on a substrate that takes care of things like intermediate-file I/O and pipelining of compiler passes. Its role is to segregate algorithms from the details of the current substrate on which they happen to run and machine to which they happen to target.

We assume that you have read the document entitled "A User's Guide to the Optimization Programming Interface" and are familiar with purpose of the data structures and functions described there. When extending the OPI, we ask that you follow the concise and consistent naming convention presented in that document so that the algorithms written using your OPI extensions are easy to understand, maintain, and debug.

In general, we extend the OPI by adding new functions based on the existing OPI data types or by adding new OPI data types and a few key associated functions. As this document illustrates, we often extend the OPI in a carefully choreographed manner so that we do not disturb or have to recompile the existing code base. You simply build new dynamically linked libraries that implement the new functionality. In this manner, we can distribute to you new implementations of our source directories and not cause you to have to re-implement any of your extension code.

Though the OPI hides the details of the underlying substrate, a person interested in extending the OPI often has to know something about the substrate. Whenever possible, we have tried to define extension mechanisms that limit your exposure to the substrate. We also, as you will see in this document, use our extension mechanism in defining several parts of our Machine-SUIF implementation of the OPI. We don't think it is fair to ask you to use an extension mechanism if we aren't willing to use it ourselves. If you are using a different implementation of the OPI, you will want to adapt these examples accordingly.

The rest of this document is organized as follows: Section 2 presents the overall structure of the code base implementing the OPI and describes how this code base is split into a hierarchy of libraries. Sections 3-6 describe example extensions associated with several different IR objects. Whenever we extend the OPI, we can classify that extension as *dependent* or *independent* of the compilation target. Section 7 discusses the development of a new target library and the concept of target-specific contexts. Section 8 shows you how you can use contexts to extend an existing target library while adding and overriding only those target-specific functions that are new or changed. We conclude in Section 9.

2 Structure of the OPI

The OPI consists of data structures built directly on top of substrate data structures and data structures implementing functionality not available in the substrate. You manipulate these data structures using class methods or plain old functions. For the purposes of adding new functionality to the OPI, you simply define a new base class or function in what we refer to as an *interface* library. The Machine-SUIF Machine, Control Flow Graph, and Bit-Vector Dataflow libraries are examples of interface libraries. Analysis and optimization passes based on the OPI refer only to the definitions in these interface libraries.

If the new functionality is independent of the compilation target, the implementation of that functionality appears directly in the `.cpp` files of the interface library. On the other hand, if the new functionality depends upon the particulars of the compilation target, each target library (e.g., the `alpha` or `x86` library) includes code to provide the appropriate functionality. Section 5 provides examples of both kinds.

If the compiler system implementing the OPI targets only a single machine, the single target library may directly specify the implementation of these target-dependent classes and functions. In a compiler like Machine SUIF, we support multiple different targets, and thus the connection of the multiple target-dependent classes and functions to the OPI interfaces is a bit more tricky. Section 7 describes how this connection is accomplished.

3 New Operand Kinds

Since operands are a key component of any machine-level IR, one would expect that the addition of a new kind of operand would require the generation of new data structures in the underlying substrate. In general, this is true, and in a moment, we provide an outline of what is involved in this tedious process. We do not however provide a lengthy explanation since we do not believe that this is something that a large percentage of our users will want to do. The OPI, especially under Machine SUIF, already comes with a fairly extensive set of atomic operand kinds.

On the other hand, we do see a need for our users to be able to define new kinds of address expressions, and thus most of this section describes a pair of simple mechanisms for supporting new address-expression operands. Extending the OPI to support these new address expressions is straightforward and does not require any knowledge of the underlying substrate.

Substrate extensions. In general, the way you would add new operand kinds to the OPI depends greatly upon the underlying substrate and the implementation of the `Opnd` class. In Machine SUIF, the class `Opnd` simply wraps a pointer to an object of the SUIF class `IrOpnd`. To define a new kind of operand, you would write a hoof specification for a new operand class derived from the abstract class `IrOpnd`, as is done for all of the atomic operand kinds currently available. (Appendix A discusses hoof and its relationship to SUIF.) You would then provide OPI functions to create, identify, and access the components of this new class. If instances of this class were immutable, you would want to implement an OPI class like `AddrExpOpnd`, which is a derived class of `Opnd`.

New address expressions. Aw we stated in *The OPI User's Guide* [2], address-expression operands are implemented as specializations of the base class `AddrExpOpnd`, which is itself a derived class of `Opnd`. If you so desired, you could implement a new kind of address expression simply by constructing an instance of the base class `AddrExpOpnd` and using its sequence methods to inspect and manipulate the suboperands of your new address expression. We do not however recommend this approach.

Instead, we recommend that you derive a subclass of `AddrExpOpnd` and define public methods in this subclass that provide more descriptive names for the suboperands. The class `BaseDispOpnd` is an example of this approach.

4 ⟨*class* `BaseDispOpnd` 4⟩≡

```
class BaseDispOpnd : public AddrExpOpnd {
  public:
    BaseDispOpnd(Opnd base, Opnd disp, TypeId deref_type = 0);
    BaseDispOpnd(const Opnd&);
    BaseDispOpnd(Opnd&);

    Opnd get_base() const;
    void set_base(Opnd);
```

```
    Opnd get_disp() const;
    void set_disp(Opnd);
};

bool is_base_disp(Opnd);
```

BaseDispOpnd defines an address expression with exactly two suboperands: the register or variable-symbol base and the integer immediate displacement. The only other declaration that you need to add to the OPI to begin programming with this new operand kind is the prototype for the predicate function is_base_disp. This OPI function distinguishes base-plus-displacement operands from the other kinds of operands in the OPI.

Notice that the class BaseDispOpnd does not add any new storage. All of the necessary storage for this address expression is provided by the base class AddrExpOpnd, which gets its storage from the SUIF class OpndAddrExp via the creator function opnd_addr_exp. You don't have to understand exactly how this storage is allocated to build a new operand kind like BaseDispOpnd. Just understand that the src sequence in AddrExpOpnd provides the storage for your custom class, and that your custom class hides how (in this case) base and disp map onto the suboperands of an instance of AddrExpOpnd.

5 ⟨BaseDispOpnd *implementation* 5⟩≡

```
bool
is_base_disp(Opnd opnd)
{
    return get_kind(opnd) == opnd::BASE_DISP;
}

BaseDispOpnd::BaseDispOpnd(Opnd base, Opnd disp, TypeId deref_type = 0)
    : AddrExpOpnd(opnd::BASE_DISP, deref_type)
{
    set_src(0, base);
    set_src(1, disp);
}

BaseDispOpnd::BaseDispOpnd(const Opnd &opnd) :
    AddrExpOpnd(is_base_disp(opnd) ? opnd : opnd_null())
{ }

BaseDispOpnd::BaseDispOpnd(Opnd &opnd) :
    AddrExpOpnd(is_base_disp(opnd) ? opnd : opnd_null())
{ }

Opnd
BaseDispOpnd::get_base() const
{
    return get_src(0);
}

void
BaseDispOpnd::set_base(Opnd opnd)
{
    set_src(0, opnd);
}

Opnd
BaseDispOpnd::get_disp() const
{
```

```
    return get_src(1);
}

void
BaseDispOpnd::set_disp(Opnd opnd)
{
    set_src(1, opnd);
}
```

Because we do not define any new storage when declaring a new specialization of `AddrExpOpnd` and an `AddrExpOpnd` is the exact same size as an `Opnd`, we can pass around or store an instance of `BaseDispOpnd` (or any other specialization of `AddrExpOpnd`) as an `Opnd`. This invariant is important because OPI functions like `set_src(Instr *mi, int pos, Opnd o)`, which replace the source operand at location `pos` in an instruction's source operand sequence with the operand `o`, do not have a variant that takes a `BaseDispOpnd`. We provide (and need) only a variant that takes an `Opnd`. You don't have to change any existing OPI functions involving operands when you add a new operand kind.

This storage constraint invariant also enables us to provide trivial constructors from an `Opnd` to a `BaseDispOpnd`. You should provide similar constructors in your custom class. The following example shows why this is important. Suppose you wanted to operate on the 0th operand of an instruction `instr`, but only if it were a base-plus-displacement address expression. You could write code something like the following:

```
Opnd opnd = get_src(instr, 0);
if (BaseDispOpnd bdo = opnd) {
    Opnd base = bdo.get_base();
    // ...
}
```

This code works because you can build a `BaseDispOpnd` from an `Opnd` if the original operand was actually of type `BaseDispOpnd`. If not, the copy constructor returns a null operand. A null operand, if evaluated in a Boolean context, evaluates to false. All other operands evaluate true.

4 Custom Notes

If you want to add a new annotation to the OPI, you should first check to see if the value sequence associated with an existing annotation satisfies your needs. For example, if you need a new kind of flag annotation, you simply define a new `NoteKey` and use the existing creator function `note_flag`. Similarly, you can simply use the value sequence produced by `OneNote<ValueType>` or `ListNote<ValueType>` for new singleton or list annotations.

Life gets more interesting if you want to define a new kind of custom annotation, but not all that much more difficult. In fact, the steps involved in the design of a new custom annotation exactly parallel the steps involved in the design of a new address-expression operand (presented in the previous section). The class `LineNote` serves as an example.

6 ⟨*class* `LineNote` 6⟩≡
```
class LineNote : public Note {
 public:
    LineNote() : Note(note_list_any()) { }
    LineNote(const LineNote &other) : Note(other) { }
    LineNote(const Note &note) : Note(note) { }

    int get_line() const        { return _get_c_long(0); }
    void set_line(int line)     { _replace(0, line); }
    IdString get_file() const   { return _get_string(1); }
    void set_file(IdString file){ _replace(1, file); }
};
```

`LineNote` is a derived class of `Note`, and as in the case with derived classes of `Opnd`, the custom classes of `Note` do not add any new storage. They get all of the storage they need from the base class. This allows us to use the type `Note` for IR components of a more derived type; we do not have to extend any existing OPI functions involving the class `Note` when adding a new custom class. And, it makes the definition of copy constructors from IR components of type `Note` trivial, as shown in ⟨*class* `LineNote` 6⟩.

The underlying storage for the base class is provided by the function `note_list_any`. This function is used only by extenders; it should never appear in the code for analyses or optimizations. This function produces a special kind of list annotation with a potentially heterogenous value sequence. In particular, each value in the sequence can be one of the following types: `long`, `Integer`, `IdString`, or `IrObject*`.

As in the case of a new address-expression operand, we define methods of the new custom annotation class that reflect the meaning of each sequence value. The base class `Note` provides protected sequence methods that allow you to inspect and change individual values as well as manipulate the sequence itself. In the `LineNote` example above, we have defined two sequence values: a line value of type `int` and a file name value of type `IdString`. The line value happens to occupy the 0th location of the value sequence, and it is accessed using the protected method `Note::_get_c_long`. We use this particular method because we know the type of the value stored in the 0th location. We can make similar statements about the file name value and methods. Please see *The OPI User's Guide* [2] for a discussion of how to program with custom annotations.

Substrate extensions. You can also add new kinds of annotations by creating new data structures in the underlying substrate. As with operands, we do not expect that you'll need to do this very often (if at all). If you do need something that cannot be satisfied with custom annotations, we suggest that you review the documentation that came with your `machine` library. It will explain how the base class `Note` was constructed on top of the substrate data structures. You can use this as a model for your new kind of annotation.

5 New Instructions

All machine instructions are of type `Instr*`. In the OPI, we carefully hide the implementation of the `Instr` class so that analyses and optimizations written using the OPI are not dependent upon any particular implementation of it. As we have stated previously, this means that you can completely change the implementation of the class `Instr`, and as long as your new implementation adheres to the OPI, you will not have to rewrite any existing optimizations. This organization provides you with a lot of freedom; you can completely restructure the `Instr` class since any changes you make are hidden behind the OPI's functional interface.

In general, there are two basic questions that you must ask when considering support for a new kind of instruction:

1. Does the class `Instr` or any of its existing derived classes contain enough of the right kinds of storage to act as the container class for your new instruction kind?
2. Will the identification or manipulation of an instance of your new instruction kind require target-specific information?

Storage considerations. To support a new instruction kind, some IR data structure in the system must contain enough storage to describe all of the instruction components. This issue is obviously dependent upon the particular implementation of the OPI that you're using. For an implementation like Machine SUIF, you will probably not need to make any changes to `Instr` or its derived classes. We provide just a few very general derived classes; each of which is capable of describing a wide range of instructions.

For example, the class `InstrAlm`[1] is capable of describing any arithmetic, logical, or memory instruction consisting of zero, one, or more source ("argument") or destination ("result") operands. You can even use this class to implement support for an architecture with predicated instructions. You could choose to implement a predicated add instruction by defining the first source operand of every instruction as the input predicate. Optimization passes not concerned with the difference between the input predicate and the rest of the source operands continue to use the sequence functions `get_src` and `set_src`, while those specifically tailored for predicated architectures could use a set of sequence functions that treated the first source operand specially.

Once you have identified (or defined) an appropriate storage container for your new instruction kind, you next must provide a way for the OPI user to create new instances of your instruction kind. We refer to functions in the OPI that create new instances of objects as "creation functions", or simply *creators*. A machine instruction creator takes an opcode and possibly some operands or other instruction components, and it produces a machine instruction.

Continuing with our predication example, if you choose to use the `InstrAlm` class as your storage for predicated instructions, you could simply use the existing `new_instr_alm` creator functions for your new instruction kind. For convenience, you may decide to implement a new creator function that distinguishes the input predicate from the rest of the source operands. The following illustrates one possible way to do this:

```
Instr*
new_instr_predicated_alm(Opnd dst, int opcode, Opnd predicate,
                         Opnd src1, Opnd src2)
{
    Instr *mi;
    mi = new_instr_alm(dst, opcode, predicate, src1);
    set_src(mi, 2, src2);
    return mi;
}
```

For more information about support for machine instructions and instruction creators, we refer you to *The Machine-SUIF Machine Library* document [1].

[1] This class is not in the OPI. It is simply how we have currently chosen to implement the part of the OPI that supports for instructions.

Target dependence. Once you have a data structure capable of describing your new instruction kind and a mechanism to create an instance, you need to define OPI functions to inspect and potentially manipulate the instruction's components. Often, you can reuse existing OPI functions for this purpose, as we did above in our discussion of instruction creators. The new issue that we want to raise here is one of target dependence. This issue is best illustrated using a simple example from Machine SUIF.

The OPI provides instruction predicates–Boolean functions that return true if an instruction is a particular kind of instruction. The functions `is_mbr` and `is_cmove` are two example predicates. The former identifies multi-way branches, while the latter identifies conditional moves. Under Machine SUIF, the implementation of the former is target independent, while the later is target dependent.

9a ⟨*function* `is_mbr` 9a⟩≡
```
bool
is_mbr(Instr *instr)
{
    return has_note(instr, k_instr_mbr_tgts);
}
```

9b ⟨*function* `is_cmove` 9b⟩≡
```
bool
is_cmove(Instr *instr)
{
    return dynamic_cast<MachineContext*>(the_context)->is_cmove(instr);
}
```

The function `is_mbr` determines whether an instruction is a multi-way branch by testing for the existence of an annotation that appears only on multi-way branches. In Machine SUIF, this annotation must appear on all multi-way branches and is independent of the target.

The function `is_cmove` returns true if the instruction is a conditional move. We explain the implementation of this function in Section 7. Here, we simply state that this function looks up the correct implementation of this predicate in the target `Context`, which was loaded based on the target annotation during the call `focus(FileBlock*)` in `do_file_block` in the pass's file `suif_pass.cpp`.

Other implementations of the OPI may handle these issues in a different manner.

6 New Execution Bodies

Every now and then, you may find that you need to extend the system with a new kind of `AnyBody`. An `AnyBody` is an `IrObject` that can be used as the body of an `OptUnit`. An `InstrList` is an example of an `AnyBody`. In this section, we review the major steps involved in building a library for a new kind of `AnyBody`. In particular, we describe our implementation of the `Cfg`, a derived class of `AnyBody`, and the structure of the control-flow graph (CFG) library.

The file `cfg/cfg.h` lists all of the important header files in the CFG library.[2] This is the header file that users of your new functionality include in their analysis and optimization passes. This section discusses the implementation of the CFG library and not just in its OPI interface.

The following list briefly describes each of the header files included in `cfg.h`. For a detailed discussion of the contents of these files, we refer you to the Machine-SUIF CFG library document.

[2]Please note that not all of the declarations in the `cfg` header files are OPI-visible functions and data structures.

- cfg_ir.h and cfg_ir_factory.h define new hoof-generated (i.e., SUIF-substrate-specific) classes and other data structures. This is where class Cfg and CfgNode are declared since they are extensions of data structures in the SUIF substrate.
- graph.h defines the set of OPI functions available for creating, inspecting, and manipulating the graph that is the CFG.
- node.h defines the set of OPI functions available for creating, inspecting, and manipulating a CFG node and its edges and contents.
- util.h contains a number of OPI helper functions that do not fall neatly into either graph.h or node.h.
- init.h lists the parts of the CFG library that you must initialize before you can start using its facilities.

To define a new AnyBody, you would create one or more substrate classes and one or more creator functions, as done in cfg_ir.hoof. You would also write an appropriate set of initialization routines as we have done in init.h and init.cpp for the SUIF substrate. The content of the remaining header files is specific to the kind of AnyBody you define. There is one piece of these header files that is important to include in any AnyBody, and that piece deals with the issue of conversion from one AnyBody to another.

The CFG library provides a creator function new_cfg that takes an InstrList and produces a Cfg. The Machine Library defines a function to_instr_list that takes an AnyBody and produces an InstrList. The class AnyBody is the base class for InstrList, Cfg, and other new OptUnit body types. It is an abstract class defining the method to_instr_list, which we use to implement the OPI function to_instr_list. In plain English, InstrList is the default form of an OptUnit body, and as such, we make it easy for you to provide a pair of converters between the new AnyBody and InstrList.

Given a pointer to the body of an OptUnit, you can dispatch based on the type of the body by using the is_a<BodyType> template predicate function, where BodyType is the derived type of the AnyBody you are interested in. For example, the following code ensures that the body of an OptUnit is a Cfg.

```
OptUnit *unit;
AnyBody *orig_body = get_body(unit);
if (!is_a<Cfg>(orig_body)) {
    // conversion
    InstrList *instrlist = to_instr_list(orig_body);
    Cfg *cfg = new_cfg(instrlist);

    // replace original body with new cfg
    copy_notes(orig_body, cfg);
    set_body(unit, cfg);

    // clean up
    delete instrlist; delete orig_body;
}
```

In order for the code above to work within a Machine-SUIF pass, you must have linked the pass with the library corresponding to the type of the orig_body and called that library's initialization function. This approach is unacceptable for a system where a new AnyBody may be defined and used long after an optimization pass is written.

As such, we define two converter passes with each kind of AnyBody we define. In the CFG library, we define a pass il2cfg that converts OptUnit bodies in InstrList form to Cfg form, and a pass cfg2il that performs the opposite conversion. Each optimization pass then simply verifies that the input `OptUnit` body is in the expected form; it is the responsibility of the person stringing the passes together to insert converter passes where necessary.

7 Target Libraries and the Context Object

A *target library* implements all target-dependent OPI functions and initializes all target-dependent OPI data structures. The `alpha` and `x86` libraries distributed with Machine SUIF are examples of target libraries. In some compilers, you specify the target library when the compiler is compiled. In Machine SUIF, the target is not specified until the compiler is run. Machine SUIF dynamically loads the required target library when it runs.

This section briefly describes the structure of a target library and the method by which we support the dynamic linking of target libraries in Machine SUIF. The key to this process is the `Context` data structure. In Section 8, we show how this data structure also supports architectural investigations by allowing you to reuse large portions of an existing target library and override only those portions you need.

7.1 Development of a target library

A target library contains `enum` definitions, target functions, and target descriptions. A target library also includes initialization code that the underlying substrate calls before using any of the target library's facilities (e.g., see the `init` modules in the Machine-SUIF `alpha` library).

The most important `enum` is probably the opcode enumeration. With two exceptions, the opcode of a machine instruction is meaningful only when interpreted with respect to a target machine. The exceptions are `opcode_null` and `opcode_label`, which are the opcodes used for the target-independent null or label instructions, respectively. The first target-specific opcode number therefore has the value 2. The particular numbers used for the target-specific opcodes are not dictated by the target instruction set architecture; they are arbitrarily assigned small integers, useful for accessing instruction properties by table lookup. The same integer that represents `mov` in an $x86$ instruction may be used for `addq` in an Alpha instruction.

For each target-dependent OPI function, a target library must provide a function with the same prototype. For example, the OPI function

```
bool is_cmove(Instr*);
```

is implemented for the Alpha platform by the `alpha` library function

```
bool is_cmove_alpha(Instr*);
```

Similarly, for each OPI class that encapsulates a piece of the target description, the target library must provide a specialization of that class. For example, the `x86` library declares `RegInfoX86` as a subclass of the OPI class `RegInfo`. For a detailed description of the purpose of the function `is_cmove` and the class `RegInfo`, we refer you to *The Machine-SUIF Machine Library* document [1].

As illustrated above, our naming convention appends the library name to the end of an OPI function to obtain the name for that function's target-dependent implementation and to the end of an OPI class to obtain the name for that class's target-dependent specialization. Machine SUIF however does not use these names to bind an OPI definition to its target-specific implementation. That is the task of the `Context` object, as explained in the next section.

7.2 The target context

A *context* gathers all the items that characterize the target machine into one record. The context data structure is a powerful tool for OPI extenders. OPI programmers, on the other hand, don't directly access the context data structure. It is constructed automatically during the (non-OPI) call to `focus(FileBlock*)`. OPI programmers access target characteristics through global functions and through the data structures that they return. Function `target_regs`, which returns a `RegInfo` pointer, is typical. The ⟨*function* `is_cmove` 9b⟩ shown earlier is another example.

We begin with a description of how contexts work in a system like Machine SUIF and thus how you use them to provide support for a new target. We then explain the reasons why we chose this design.

Building and working with `the_context`. Let's begin with a look at the implementation of ⟨*function* `is_cmove` 9b⟩ in greater detail. This function is declared and defined in the `machine` library, the main interface library in Machine SUIF. It dynamically casts the global variable `the_context` to `MachineContext*` and makes an `is_cmove` method call.

Digging deeper, we see that the declared type of `the_context` is `Context*`. The class `Context`, however, is just an empty base class, as shown below. The key feature of this type is that it is known to everyone, machine dependent and independent.

12a ⟨*class* `Context` 12a⟩≡
```
class Context {
  public:
    Context() { }
    virtual ~Context() { }
};
```

The `is_cmove` method is actually declared in a different *interface* class, called `MachineContext`. (The `target_regs` method is declared here too.) Interface classes are declared in interface libraries along with their OPI functions.

It is important to note that interface classes are not subclasses of `Context`. Each interface class is simply a chunk of interface that target libraries use when composing an actual context instance that is right for the target in question. The method of composition is multiple inheritance. So while the declared type of `the_context` is `Context*`, it points to an object whose class inherits not only from `Context`, but also from several interface classes.[3] The context object for the `alpha` library is declared as follows:

12b ⟨*class* `AlphaContext` 12b⟩≡
```
class AlphaContext : public virtual Context,
                     public virtual MachineContextAlpha,
                     public virtual SuifVmContextAlpha
{ };

#endif /* ALPHA_CONTEXT_H */
```

[3] Context-interface classes are named for the libraries that define them: `MachineContext` is defined in the `machine` library, `SuifVmContext` in the `suifvm` library, and so on.

The actual interface classes used in the declaration of `AlphaContext` are derived classes of the base interface classes found in the interface libraries. To reiterate, the `machine` library is an interface library; the context interface that it defines is `MachineContext`. The `alpha` library is an implementation library; the context class that it defines realizes a number of context interfaces, including `MachineContext` (through the derived class `MachineContextAlpha`).[4] The implementation of `MachineContextAlpha::is_cmove` simply invokes the target-specific function `is_cmove_alpha`.

The `init.cpp` file in the target library contains the last piece of this puzzle. The library's initialization code takes responsibility for generating the concrete-context pointer that gets stored in `the_context` by registering a function in `the_context_creator_registry` map. For the `alpha` library, this context-creator function invokes the `AlphaContext` constructor to create a new concrete-context object.

Multiple inheritance as an implementation technology. So why not call the methods of `the_context` directly? For one thing, it's shorter to write calls on global functions, and these things occur quite frequently. More importantly, the context is not needed in some optimization settings, such as run-time optimization. So for portability, code written against the OPI should be almost "context free". In other words, we use object-oriented techniques to support the extensibility goals of Machine SUIF, but hide this technology behind an OPI based largely on functions for portability.

Ok, but why are concrete contexts in Machine SUIF built using multiple inheritance? Because it allows more flexibility in combining libraries than traditional linear inheritance. To see why, consider the following argument. The interface libraries used by an optimization pass are known statically. Its author chooses only those that the pass needs. The target library used by the pass, however, is chosen and linked dynamically, based on the target associated with the file being compiled. How then can there be enough agreement between the context class known when the pass is compiled and the one generated when the target-specific library is linked in? If contexts are derived linearly, then the statically-known class cannot (in general) skip a level of derivation and still be compatible with the dynamically-generated context object.

Multiple inheritance from independent context-interface classes, together with dynamic casting based on run-time type information (RTTI), solves this problem cleanly. A target library creates a context that contains the union of all the information needed by passes that will use it. A given pass then accesses only those parts of the context that concern it.

8 Target Library Extension

Though it is not difficult to build a target library from scratch, you don't want to do this for multiple different revisions of the same instruction set architecture (ISA), and you certainly don't want to do this for different implementations of the same ISA. As such, we provide you with the ability to refine an existing target library. This approach is especially advantageous for experimenting with new architectural features (e.g., new instructions or new register-file properties).

This section describes the implementation of the `x86_ppro` target library. This library is a refinement of the `x86` target library; it adds support for instructions defined in the Pentium and Pentium Pro. We begin with an illustration of how to extend the opcode enumeration. The new instructions require us to override the implementations of the OPI functions `target_printer`, `is_cmove`, and `target_code_gen` from the `x86` library. We conclude with a review of the library's initialization code.

[4]The `suifvm` library is exceptional; it plays both roles. It introduces the machine-independent `SuifVmContext`, but it also defines a concrete context that is specific to the SUIFvm target.

8.1 Opcode extensions

The Intel Pentium and Pentium Pro microprocessors add several new instructions to the x86 ISA. To extend the opcode enumeration defined in the `x86` target library, we number the new opcodes starting from one beyond the value of the C preprocessor variable `LAST_X86_OPCODE`. We define 47 new opcodes, of which `WRMSR` is the last.

14a ⟨*opcode* `WRMSR` 14a⟩≡

```
const int WRMSR    = LAST_X86_OPCODE + 47;

#undef LAST_X86_OPCODE
#define LAST_X86_OPCODE x86::WRMSR
```

To allow for extensions of this extension library, we re-define `LAST_X86_OPCODE` to the value of this constant.

The file `opcodes.cpp` extends two target-specific tables `x86_opcode_names` and `x86_invert_table` that associate a string name and the complementary branch condition (if any) with each opcode. Since the Pentium Pro does not add any new conditional branches, the function `init_x86_ppro_invert_table` sets each new table entry to an invalid opcode value. These routines are invoked in the library initialization ⟨*function* `init_x86_ppro` 16⟩.

8.2 Extension contexts

The inclusion of the new instructions requires us to extend the `x86` printer class, `PrinterX86`, and override the OPI function `target_printer`. Some of these instructions are conditional moves, and thus we want to override the `x86` implementation of the OPI function `is_cmove`. Finally, we would like to take advantage of the new floating-point compare operations that allow us to directly update the `eflags` register. This means that we would like to override the OPI function `target_code_gen` that provides us with a code generator from SUIFvm instructions to instructions of the target architecture. The next subsection presents a recipe for extending the target-description objects. In this subsection, we show how you can combine these extensions with pieces from the `x86` library to produce an `X86PProContext` object that invokes the appropriate OPI functions.

14b ⟨*class* `X86PProContext` 14b⟩≡ (14c)

```
class X86PProContext : public virtual Context,
                       public virtual MachineContextX86PPro,
                       public virtual SuifVmContextX86PPro
{ };



#endif /* X86_PPRO_CONTEXT_H */
```

Like the `X86Context`, `X86PProContext` inherits from `Context` and derived classes of `MachineContext` and `SuifVmContext`. As shown below, the derived classes are just specializations of the `x86` versions of these interface classes. The specializations simply override the methods corresponding to the OPI functions that we wish to override.

14c ⟨*class* MachineContextX86PPro 14c⟩≡
```
class MachineContextX86PPro : public MachineContextX86 {
  public:
    Printer* target_printer() const;

    bool is_cmove(Instr*) const;
};
```

⟨*class* X86PProContext 14b⟩

15a ⟨*class* SuifVmContextX86PPro 15a⟩≡
```
class SuifVmContextX86PPro : public SuifVmContextX86 {
  public:
    CodeGen* target_code_gen() const;
};
```

The implementations of the `target_printer` and `target_code_gen` methods either build a new instance of a `PrinterX86PPro` or `CodeGenX86PPro` object, respectively, or return cached versions of the appropriate object. The implementation of the `is_cmove` method is shown below:

15b ⟨MachineContextX86PPro::is_cmove 15b⟩≡
```
bool
MachineContextX86PPro::is_cmove(Instr *mi) const
{
    return (is_cmove_x86_ppro(mi) || is_cmove_x86(mi));
}
```

Notice that the implementation assumes that the target-specific functions are independent. The function `is_cmove_x86_ppro` does not re-implement any of the functionality implemented by `is_cmove_x86`; it identifies only those new instructions that are conditional moves and relies on code in the `x86` library to deal with previously-defined instructions. As discussed next, we adhere to this kind of an approach for extending even the most complicated of target-specific objects.

8.3 Extension of target-description objects

OPI functions like `target_printer` and `target_code_gen` return pointers to class objects that describe or produce target-specific information. Since all of these objects are extended in the same basic way, we focus on extensions of the `Printer` class here. The `x86_ppro` library also contains code demonstrating how you could extend the `CodeGen` class.

All of the new `x86_ppro` instructions are classified as `alm` instructions, and thus the class `PrinterX86PPro` simply has to override the method `print_instr_alm` of `PrinterX86`.

15c ⟨*class* PrinterX86PPro 15c⟩≡
```
class PrinterX86PPro : public PrinterX86 {
  protected:
    virtual void print_instr_alm(Instr*);

  public:
    PrinterX86PPro();
};
```

```
#endif /* X86_PPRO_PRINTER_H */
```

The constructor for this class extends target-specific table `print_instr_table` with a pointer to the appropriate print method for each new opcode. The public method `print_instr_alm` first checks to see if the opcode of the instruction to print is one of our new opcodes or not. If it is, we specify how to print the instruction in the implementation of this method. If it is not, the routine simply invokes the base class method `PrinterX86::print_instr_alm`.

8.4 Library initialization

The module `init.cpp` contains the library initializations that the underlying substrate must perform before it or any of the optimization passes can use the `x86_ppro` target library's facilities.

16 ⟨*function* `init_x86_ppro` 16⟩≡

```
extern "C" void
init_x86_ppro(SuifEnv *suif_env)
{
    static bool init_done = false;

    if (init_done)
        return;
    init_done = true;

    init_machine(suif_env);
    init_suifvm(suif_env);
    init_x86(suif_env);

    k_x86_ppro = "x86_ppro";

    // initializations for opcode-indexed tables
    init_x86_ppro_opcode_names();
    init_x86_ppro_invert_table();

    the_context_creator_registry[k_x86_ppro] = context_creator_x86_ppro;
}
```

The `x86_ppro` library requires the `machine`, `suifvm`, and `x86` libraries to be loaded. The function `init_x86_ppro` then initializes a string constant and a few of the opcode-indexed tables. The target-description objects are not created until they are first needed. Finally, the context-creator function for the `x86_ppro` context is registered with the system. This function simply invokes the constructor for `X86PProContext`.

9 Summary

In this document, we have discussed many of the most common kinds of system extensions, and we have presented actual examples of these extension mechanisms in action in our distributed code. This document is not meant to be an exhaustive listing. However, it does illustrate the philosophy and general recipe behind our support for extensions (and improvements) to the OPI in general and Machine SUIF in particular. For more details on the items presented here, we refer you to the library documents that came with your system.

10 Acknowledgments

This work was supported in part by an DARPA/NSF infrastructure grant (NDA904-97-C-0225), a NSF Young Investigator award (CCR-9457779), and a NSF Research Infrastructure award (CDA-9401024). We also gratefully acknowledge the generous support of this research by Advanced Micro Devices, Compaq, Digital Equipment, Hewlett-Packard, International Business Machines, Intel, and Microsoft.

References

[1] G. Holloway and M. D. Smith. *The Machine-SUIF Machine Library.* The Machine-SUIF documentation set, Harvard University, 2000.

[2] G. Holloway and M. D. Smith. *A User's Guide to the Optimization Programming Interface.* The Machine-SUIF documentation set, Harvard University, 2000.

A SUIF's Hoof Specification

The SUIF group at Stanford uses a notation called *hoof specification* to describe the structure of the C++ classes that represent the SUIF IR and the relationships between those classes. The C++ IR data types are generated automatically from hoof specifications. Users learn what methods are created from the hoof descriptions, and they program in terms of those methods. To give you a flavor, here is a partial excerpt from the `machine_ir.hoof` specification in the `machsuif2/machine` directory:

```
abstract IrOpnd : SymbolTableObject {
    virtual int kind;
    virtual Type* reference type;
};

concrete OpndVar : IrOpnd {
    VariableSymbol* reference var;

  public:
    int   get_kind() const { return opnd::VAR; }
    Type* get_type() const { return _var->get_type(); }
};
```

This declares two hoof classes: `IrOpnd` and `OpndVar`. `IrOpnd` is abstract in the C++ sense: it has no instances of its own; rather, all of its instances are instances of subclasses. It also has no fields that are strictly its own, but it has two "virtual" fields: an integer defining the operand kind and a reference to a `Type` object. These virtual fields indicate that the C++ class produced from this Hoof specification should contain four virtual methods: `get_kind`, `set_kind`, `get_type`, and `set_type`. The implementation of these virtual methods is left to the subclasses of `IrOpnd`.

`OpndVar` is a concrete subclass of `IrOpnd`. In addition to the fields defined by `IrOpnd`, `OpndVar` declares a field that refers to a `VariableSymbol`. This field consumes real storage in an instance of `OpndVar`. This declaration also produces methods that allow a user of an `OpndVar` object to get and set class variable `var`. Finally, the hoof specification of `OpndVar` specifies the implementation of the virtual methods `get_kind` and `get_type`. Since variable-symbol operands are atomic in the OPI, we do not need to provide any implementations of the set methods.

Please see the Stanford SUIF documentation set for further information concerning hoof specifications and the development of new SUIF classes.

B Header-File Inclusion and Library Initialization

When extending Machine SUIF, you may want to follow the conventions for header-file inclusion and library initialization that we have used in building the system.

The first guiding principle is that each source file should record its own explicit dependences by `#include`-ing appropriate header files. And it should never have to cater for the dependences of files that it includes. Similarly, each library or pass should call the initialization function for any library that it depends on, and not rely on knowledge that one library may initialize another.

A second principle is that the exports of a library should be encapsulated in a single header file for the library, so that no client needs to be aware of the internal structure of a library that it depends on.

Library-exports header file. For example, clients of the CFG library use this directive:

```
#include <cfg/cfg.h>
```

Note that we use angle brackets, rather than quotation marks, to delimit the file path. This is not essential, but it acts as a reminder that all header files are installed under `$NCIHOME/include`, and it helps to prevent ambiguities arising because different C++ implementations have different rules for deciding where to search for headers included in headers.

The path for a library-exports header file always has the above form, i.e., it uses the library name as both a directory and as the stem of the file name.

A library header file contains only `#include` directives. The header files that they inject declare the exports of the library. Take the header file for the BVD library for example:

18 ⟨*contents of* `bvd.h` 18⟩≡

```
#include <bvd/flow_fun.h>
#include <bvd/catalog.h>
#include <bvd/solve.h>
#include <bvd/liveness.h>
#include <bvd/reaching_defs.h>
#include <bvd/init.h>
```

There's no need to include header files from other libraries (such as CFG) in `bvd.h` because any such dependences are handled within the local headers themselves. (E.g., `solve.h` includes `cfg.h`.)

Header inclusions in other source files. Each `.h` and `.cpp` file implementing a library or pass `#include`s all the header files that the compiler needs to see to compile it properly. When it needs exports of (another) library, it includes the exports header of that library, using the angle-bracket notation shown in the examples above. But a `.cpp` file within a library never includes that library's own exports header. Instead it includes just the local headers that it needs, using the `<...>` notation and the directory qualifier, just like in the `.h` files.

Library initialization. The rule for deciding which library initialization routines (those named `init_`*library*) to invoke when initializing a library or pass parallels the rule for header-file inclusion.

If there's a need to explicitly `#include` the exports header of a library, then you should explicitly invoke its `init_`*library* routine, and otherwise not.

Link specification. It would be nice if the library search list submitted to the linker, represented in a `Makefile` by using a `LIBS` definition, could exactly parallel the other explicit mentions of a library. Implementing that seems more trouble than it's worth, however. So we just let the linker tell us via error messages what implicitly-required libraries to include in each `LIBS` list.

Substrate-boundary library. The `machine` library is special in that it serves as a boundary layer between Machine SUIF and the base SUIF substrate. In general, all dependences on base SUIF are captured in `machine/machine.h` and in an additional header file called `machine/pass.h` that each pass should include in its "SUIF wrapper" implementation files. Likewise, the `init_machine` function that sets up the `machine` library also takes care of initializing libraries imported from base SUIF.

C More Naming Conventions

The *OPI User's Guide* describes the conventions used in forming OPI identifiers. The Machine SUIF implementation observes those conventions, and it adds some that did not come up in specifying the OPI.

When you extend the system, you often need to make up new enumeration identifiers. New opcodes, new architectural registers, and new kinds of instrumentation all call for new enumeration symbols. For these symbols, which may have wide visibility and get a lot of use, we use upper-case identifiers, articulated with underscores. To avoid name collisions, we limited use of C++ namespaces, and we use the name of the current library to name such namespaces themselves.

For example, the enumeration literals for opcodes in the Alpha architecture have names like `ADDQ` and `MOV`, and they are defined in namespace `alpha`. In code that uses the Alpha library, the programmer can choose to qualify names explcitly, as in `alpha::MOV`, or to open the namespace with directive

```
using namespace alpha;
```

This is not to say that we put every enumeration type definition into a namespace. Where local constant names seem helpful for the clarity of the code, we use ordinary `const` or `enum` declarations. We usually follow the long-standing C convention of upper-case symbols to distinguish constants from variables.