An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization

Release version 2.00.11.30

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

November 30, 2000

Abstract

Machine SUIF is a flexible and extensible infrastructure for constructing compiler back ends. With it you can readily construct and manipulate machine-level intermediate forms and emit assembly language, binary object, or C code. The system comes with backs ends for the Alpha and $x86$ architectures. You can easily add new machine targets and develop profile-driven optimizations.

Though Machine SUIF is built on top of the Stanford SUIF system (version 2.1), the analyses and optimizations within Machine SUIF are not SUIF specific. By rewriting the implementation of an extensible interface layer, you can easily port the Machine-SUIF code base to another compilation environment. We refer to this interface as the Optimization Programming Interface (OPI). The OPI allows us to write optimizations and analyses that are parameterized with respect to both a target and an underlying environment. We use this capability to share optimization passes between Machine SUIF and Deco, a system for dynamic code optimization.

This document introduces Machine SUIF, presents the overall rationale behind its design, illustrates how one writes a parameterized pass, and describes how to get started using the system. It concludes with a road map for the rest of the documentation that comes with Machine SUIF.

Contents

1 Introduction

Machine SUIF is a flexible, extensible, and easily-understood infrastructure for constructing compiler back ends. We built Machine SUIF with the philosophy that the "textbook" portions, i.e. the optimization and analysis passes, should be coded in a way that makes them as independent of the compiler environment and compilation targets as possible. By adhering to this philosophy, we are able to distribute both a working compiler and a collection of analyses/optimizations that can be quickly and easily inserted into a new compilation environment.

In particular, the Machine-SUIF distribution contains a working compiler based on the Stanford SUIF compiler infrastructure (version 2.1). This compiler is capable of producing optimized code for machines based on the Alpha or x86 architectures. However, the analyses and optimizations distributed in Machine SUIF do not directly reference any SUIF constructs or embed constants from any target machine. Instead, each is written using a standardized view of the underlying compiler environment, an interface layer that we refer to as the *Optimization Programming Interface* (OPI).

This document is an introduction to Machine SUIF. It briefly presents our overall design rationale, gets you started using the system, and directs you to the rest of the infrastructure documentation. We assume that you are familiar with the basic concepts of SUIF 2.1; if not, we encourage you to see the documentation on the SUIF 2 home page[1].

The next section enumerates our goals for Machine SUIF and outlines how we achieved them. Section 3 then describes how to install Machine SUIF and start using it to compile programs. Section 4 provides an overview of the general flow of our back-end compilation system and presents several example back-end scripts. Section 5 describes what's involved in developing optimization passes using our OPI, while Section 6 shows you how to extend the OPI for new optimizations. Section 7 describes the steps that you need to take in order to provide support for a new target architecture. Section 8 discusses how you can change the structure of the Machine-SUIF intermediate representation (IR) and how to port the optimizations to a new environment. Section 9 briefly outlines where to go next for more details.

2 Goals

We designed Machine SUIF for a wide range of users. In particular, we support those interested in:

- building new optimizations that are parameterizable with respect to the target machine and portable across compiler environments;
- adding support for a new or augmented target architecture;
- developing a new IR, without having to rewrite all of our optimizations.

To adequately address the needs of these users, we developed Machine SUIF with three primary goals in mind. First and foremost, Machine SUIF had to be easy to use, especially if you wanted to do something simple, and straightforward to retarget so that we could use it in architectural investigations. Second, it had to support the modular development of sophisticated optimizations. We wanted to provide our students and colleagues with the ability to contribute passes and benefit from the efforts of others. Finally, it had to built in a manner that permitted reuse of existing optimizations directly in an optimization environment with significantly different constraints. Specifically, we wanted to be able to reuse optimizations built for Machine SUIF in Deco, a project at Harvard University in *dynamic* or *on-line* optimization. The rest of this section explores each of these goals in greater detail.

[1] See URL `http://suif.stanford.edu/suif/suif2/`.

Ease of use. For those using Machine SUIF simply as a compiler, Sections 3 and 4 of this overview should be a sufficient introduction to the system.

Since we designed Machine SUIF as a research compiler, we expect that most of our users will want to add or change the system in some way. (Hey, we're never satisfied either.) Sections 5 and 6 should help you to develop new analysis or optimization passes. These sections describe how we use libraries to ease the building of Machine SUIF passes. These libraries, e.g., the control-flow and data-flow analysis libraries, provide abstractions that aid in the coding of certain kinds of optimizations. These sections also discuss the OPI and its support for the development of *parameterized passes*: parameterization separates target- and environment-specific details from the algorithms of analysis or optimization passes. The OPI also hides the implementation of our IR, many details of which are uninteresting to someone simply wanting to code a new optimization algorithm. After reading these sections, we encourage you to read our separate documentation on the OPI.

In Machine SUIF, the objects that hold target-specific information are parameters to the analysis and optimization passes. These objects, which include both functions and data structures, reside in target-specific libraries (or "target libraries" for short). Adding support for a new target architecture therefore simply requires you to code a new target library or extend an existing one. Existing optimization passes work without change for the new library. The only time you need to change an existing pass is when you want to change that pass's algorithm. By separating algorithmic details from the architectural details, we have made it easier, as described in Section 7, to experiment with new microarchitectural features.

Finally, we have tried to do the "little things" that make it easier to use someone else's software.

- We provide extensive documentation. In particular, we use Norman Ramsey's `noweb` system [9] to produce the majority of our hypertext and hardcopy documentation. This literate-programming tool lets you combine documentation and code in the same source file. We develop `noweb` files for our most important header files, and we incorporate these files into documents describing the libraries that contain them. We use simple code comments to document Machine-SUIF passes and library implementation files respectively.
- We use short, informative naming conventions in our code. When we use an abbreviation for a particular word, we are careful to use the same abbreviation everywhere. We include a glossary of our most-common abbreviations at the end of *The OPI User's Guide* [8].

Quality of optimized code. Machine SUIF is a powerful system for producing optimized code. It is modular, making it easy to add, remove, and rearrange passes. We typically develop optimization passes that focus on a single action, such as dead code elimination. Nearly all such passes can be inserted at any point in the back-end flow. Thus, the writer of an copy-propagation pass can focus on data flow and operand rewriting and assume that a subsequent run of the dead code elimination pass will remove any dead code produced during copy propagation.

Reuse of optimization code. Our research program is interested in a wide range of techniques for program optimization. Our interest spans the spectrum from traditional methods performed during compilation through hardware and software optimization techniques performed at run time. We would, for example, like to use the optimizations written for Machine SUIF in an on-line optimization system and debug/test optimizations written for the on-line optimizer in Machine SUIF.

Machine SUIF implements one realization of the OPI. Since the OPI hides the implementation specifics of Machine SUIF (i.e., the substrate), we can easily produce another realization of the OPI appropriate for an on-line optimizer. To ensure that the optimizations written for Machine SUIF are usable in this on-line optimizer, we write all of our Machine-SUIF optimizations to conform to the OPI's specification. In this way, Machine SUIF can be a flexible infrastructure (with full-featured IR objects) that is easy to extend, while an on-line optimizer can use a different infrastructure employing light-weight IR objects for efficiency. And both can use the same set of optimization codes. *The OPI User's Guide* [8] discusses this issue and its implications in greater detail.

3 Setting Up Your Environment

Machine SUIF depends on the base SUIF system, which is distributed from the Stanford SUIF 2[2] and National Compiler Infrastructure[3] (NCI) web sites.[4]

The following discussion assumes that you are using a UNIX-based operating system as your development platform. It also assumes that you know how to unpack the distributions and set up the `nci` source directories. If not, please first review the `README` files contained in the distributions. Please pay careful attention to the instructions regarding the environment settings required to compile SUIF.

3.1 Compiling Machine SUIF

You should first unpack and install the base SUIF distribution. Instructions are in `nci/README`. The same environment variables that you use for this step (e.g., `MACHINE`, `NCIHOME`, and `COMPILER_NAME`) are required for compiling the Machine SUIF distribution.

Next, unpack the Machine SUIF. You can put it wherever you like. Follow the instructions in `README.machsuif` file in the root directory of the distribution to complete the installation. Be sure to have the `MACHSUIFHOME` environment variable set as described there whenever you build or rebuild part of the Machine SUIF system.

3.2 Compiling with Machine SUIF

A machine-specific optimization must have access to information about the organization and microarchitecture of the target machine. We encapsulate this target-specific information in target libraries (e.g., the `alpha` library). During optimization, the compiler selects the appropriate target library and binds the information in this library into the current environment by constructing a `Context` object. The compiler knows which library to load by reading the library name from an annotation incorporated in each Machine-SUIF IR file. So all procedures that originated in a given source file are optimized using a single, consistent set of target-specific information.

So, how do you as a person running the compiler specify a target library? It is fairly simple. Only a few of our Machine-SUIF passes actually change the value of the `target_lib` annotation. Each of these passes has a command line parameter, `-target_lib`, that allows you to specify the name of a target library. Alternatively, if you do not specify the target library as a command line parameter, the compiler will check the environment variable `MACHSUIF_TARGET_LIB`. The command line argument overrides the environment variable, and if neither command line argument nor environment variable is defined, the compiler stops with an assertion failure.

[2] See URL `http://suif.stanford.edu/suif/suif2/index.html`.

[3] See URL `http://nci.pgroup.com/`.

[4] The **Downloads** page at the NCI site currently offers some out-of-date source files under links labeled **The base NCI system** and **Conversion code between suif1 and suif2**. Ignore those links! Take only the binary distributions of **The C front end** from the NCI site. Take the rest of SUIF from the Stanford site.

4 Back-End Organization and Flow

You can think of Machine SUIF as a machine-level IR, with instructions having type `Instr`, that plays a role like that of `Statement` or `Expression` in base SUIF. However, the Machine-SUIF infrastructure includes much more than just another SUIF IR; it augments SUIF with many new libraries and passes that make it straightforward to develop machine-specific optimizations for existing or future machine models.

For us then, the back end consists of those compiler passes that start with the lowering of the SUIF `Expression` or `Statement` IR into the Machine-SUIF IR and that end with a translation from the Machine-SUIF IR into assembly, machine, or C code. During back-end compilation, the Machine-SUIF IR for a compiled program will transition through at least two Machine-SUIF IR dialects. We begin with a brief discussion of the ideas that led us to the current IR and then move on to define what we mean by a dialect of the Machine-SUIF IR. With this as background, Sections 4.3 and 4.4 describe the general flow of a Machine-SUIF back end in more detail. This section ends with some example back-end scripts and a word about the constraints on pass orderings.

4.1 IR organization

Research on machine-specific compile-time optimizations is closely tied to research in computer architecture. To support this area of research, a compilation system must possess an IR that is both extensible and expressive. We require extensibility so that we can experiment with new compiler-visible instructions, architectural features, and hardware mechanisms. SUIF was built with extensibility as a primary design goal, and thus it has been fairly simple for us to meet this requirement. We desire expressiveness in our IR so that every single machine instruction is representable by a single IR instruction. A one-to-one correspondence between IR instructions and actual machine instructions is required for optimizations such as global instruction scheduling.

The development of an extensible and expressive IR could be a lifelong project within itself. As mentioned earlier, we would like to avoid actually spending our lives re-coding compiler passes every time we make a small change to the IR. To avoid this problem, we have defined OPI functions that create, inspect, and manipulate IR objects in a target-machine-specific manner. The OPI hides the details of the IR implementation. In other words, the OPI supports the needs of the optimization writer without revealing the actual structure of the IR. The Deco project, for example, takes advantage of this capability to use the existing Machine-SUIF analysis and optimization code with a completely new IR.

4.2 Dialects

Even though Machine SUIF supports optimization targeting many different machines, all Machine-SUIF IR files have the same binary representation. We simply interpret the contents of this IR file differently for each different target. We say that each target library defines a *dialect* of the Machine-SUIF IR.

Machine SUIF has one distinguished dialect called `suifvm`, which stands for SUIF virtual machine. This dialect is the target of the Machine-SUIF lowering pass. This pass lowers from SUIF `Expression` and `Statement` form into `suifvm` instruction form. All the code generators within Machine SUIF are code generators from `suifvm` to some specific target architecture. By defining this dialect, we are able to de-couple extensions made to base SUIF from Machine SUIF's provision of multiple target-specific back ends. If an extender of base SUIF wishes to connect to Machine SUIF, he or she need only write a lowering pass to `suifvm` . If someone wants to add a new target to Machine SUIF, he or she need only develop a new target library for Machine SUIF. The base SUIF extender and the new target developer do not need to know about each other's work.

.c → front-end → IR-restriction → optimizations → code-gen → .s

(a)

.c → front-end → code-gen → optimizations → .s

(b)

Figure 1: Two approaches that satisfy the goal of a one-to-one correspondence between IR and machine instructions. The first splits the code generation into two pieces. The second relies on either retargetable compiler technology to facilitate the quick creation of machine-specific optimization passes or abstraction to eliminate the need to re-code each optimization for each architecture.

Currently, dialects differ only in their interpretation of the integer opcode value stored in the `Instr` structure. The opcode value is the key into many of the target-specific data tables, and thus it must be interpreted correctly in order to access the appropriate machine-specific information needed for optimization.

4.3 General flow

We considered two basic approaches that satisfy the goal of a one-to-one correspondence between IR instructions and machine instructions. The approaches differ in where they perform code generation, the actual mapping from an IR opcode to a machine opcode.

The first approach postpones this mapping until the last possible moment in the compilation process (see Figure 1a). To ensure that there is a one-to-one correspondence between the IR and machine instructions, there is an earlier pass in the compilation process that restricts the IR form so that each IR instruction can be mapped directly to a single machine instruction. The machine-specific optimization passes maintain this one-to-one correspondence. The IMPACT compiler [1] uses an approach like this.

The second approach performs the mapping from IR instructions to machine instructions early in the compilation process (see Figure 1b). Machine SUIF employs this approach. The parameterization of our optimization passes keeps us from having to re-implement each of our machine-specific optimizations for each target architecture. This same idea is found in retargetable compilers, such as the vpcc/vpo compiler [2], based on compiler compiler technology. Our approach simply performs target specialization of the compiler dynamically rather than statically. We find this late binding of target-specific information useful for environments where the target description and compiler requirements change frequently.

4.4 Compilation specifics

We now describe how to use SUIF and Machine SUIF to compile a C program to Digital Alpha/UNIX assembly code. Begin by reading the directions[5] in the SUIF documentation that describe how to produce a `.suif` file from a C input file. Next, apply `SUIF` transformations that make an IR file acceptable for lowering into Machine SUIF. If your file is named `hello.suif`, for example, the following command produces a lowered SUIF version called `hello.lsf` in which some structured forms like loops and conditional statements have been replaced by less structured equivalents:

```
suifdriver -e "require basicnodes; \
               require suifnodes; \
               require cfenodes; \
               load hello.suif; \
```

[5] See URL `http://suif.stanford.edu/suif/suif2/doc/howtocompile.html`.

```
require transforms; \
dismantle_field_access_expressions; \
dismantle_structured_returns; \
compact_multi_way_branch_statements; \
dismantle_scope_statements; \
dismantle_ifs_and_loops; \
flatten_statement_lists; \
rename_colliding_symbols; \
insert_struct_padding; \
insert_struct_final_padding; \
save hello.lsf"
```

A corresponding script, called `do_lower`, comes with the Machine SUIF distribution. So instead of typing the bulky command above, you can write:

```
do_lower hello.suif hello.lsf
```

A minimal back end. A back end starts with a lowering pass; the lowering pass that accepts the output of `do_lower` is called `s2m`. The output of `s2m` is a Machine-SUIF IR file that targets `suifvm`. We supply a target library for `suifvm` that allows you to run optimization passes on a Machine-SUIF IR file of this dialect.

Typically, you'll want to translate the `suifvm` code into code for a real machine target. This translation requires two basic steps: translate `suifvm` instructions into the real machine's instructions (code generation); and translate the `suifvm` symbol and virtual-register operands into machine registers and memory locations (register allocation). A minimal Machine-SUIF back end consists of four steps related to these two basic steps: perform most of the code generation; allocate registers; finish machine-specific translation; and emit the ASCII representation of the Machine-SUIF IR file. The following commands carry out these steps to generate an Alpha assembly file:

```
do_s2m    $f.lsf $f.svm
do_gen    -target_lib alpha $f.svm $f.avr
do_il2cfg $f.avr $f.afg
do_raga   $f.afg $f.ara
do_cfg2il $f.ara $f.ail
do_fin    $f.ail $f.asa
do_m2a    $f.asa $f.s
```

The `gen` pass (called `do_gen` when run from the shell) does most of the machine-specific code generation. It does not allocate registers (beyond those necessary, for example, because a certain instruction uses a specific, implicit register). Since `gen` translates from one Machine-SUIF dialect to another (i.e., from `suifvm` to `alpha` in this case), we need to supply the name of the target library. Here, we used the command-line option method, though we could easily have set the appropriate environment variable instead. To generate code for a different target, you simply provide a different library name.

The `il2cfg` pass transforms the IR from simple instruction-list form to CFG form, which is a prerequisite for most optimization passes.[6]

The `raga` pass performs register allocation using an algorithm described by George and Appel [4]. The `fin` pass completes the translation process by laying out the stack frame for each procedure, replacing stack-allocated symbols by their stack-based effective addresses, and inserting the procedure entry and exit code sequences. These entry/exit sequences are delayed until this finalization pass so that other

[6] In particular, the `dce`, `raga` and `peep` passes expect their input in CFG form.

Machine-SUIF passes can easily reallocate registers in a procedure, for example, and not have to change the register save/restore code at that procedure's entry/exit points. Neither of these passes changes the target associated with the file being compiled.

The `cfg2il` pass reverses the transformation made by `il2cfg`: it takes in the CFG form and produces the simple instruction-list form of the IR.

The `m2a` pass translates the Machine-SUIF representation for a program into an architecture-specific ASCII assembly-language file (a `.s` file). This pass also creates the data directives for the file by translating its symbol table. Before this point, it is expected that all Machine-SUIF passes maintain the symbol table (and don't create machine-specific data directives in instruction lists). The resulting `.s` file can then be assembled by the target machine's assembler to create a `.o` file.

The value of using a separate shell command (`do_...`) for each pass and sending intermediate results through the file system is that it gives you finer control when debugging. However, you can also pass intermediate results in memory, which is considerably faster:

```
suifdriver -e \
  "require cfenodes; \
   load $f.lsf; \
   require s2m;    s2m; \
   require gen;    gen -target_lib alpha; \
   require il2cfg; il2cfg; \
   require raga;   raga; \
   require cfg2il; cfg2il; \
   require fin;    fin; \
   require m2a;    m2a $f.s"
```

A fancier back end. The code produced by the minimal back end described above will be highly unoptimized. To improve the output-code quality, you can run machine-specific optimization passes between `s2m` and `m2a`. For example, you may wish to run a dead-code elimination pass `dce` to eliminate useless operations:

```
suifdriver -e \
  "require cfenodes; \
   load $f.lsf; \
   require s2m;    s2m; \
   require il2cfg; il2cfg; \
   require dce;    dce; \
   require cfg2il; cfg2il; \
   require gen;    gen -target_lib alpha; \
                   il2cfg; \
   require raga;   raga; \
                   dce; \
                   cfg2il; \
   require fin;    fin; \
   require m2a;    m2a $f.s"
```

This back end invokes `dce` twice: once before code generation and once after register allocation. This illustrates the power of parameterized optimizations; the same pass works for all Machine-SUIF dialects.

The `src/machsuif/README.machsuif` file included with your distribution of Machine SUIF describes the optimization passes available.

Viewing the results of intermediate stages. A script called `do_print` distributed with Machine SUIF allows you to print the contents of an intermediate file in a form resembling assembly language, even if it is not actually ready for assembly. Suppose for example that you want to look at the output of the code-generation stage. If the intermediate file is called `hello.avr`, then

```
do_print hello.avr hello.avr.txt
```

produces a text file `hello.avr.txt` expressing instructions in assembler syntax. Since the IR has yet to undergo register allocation, the listing contains virtual registers (e.g., `$vr42`) and also symbols used where the assembler would expect proper machine-register designators.

If you omit the second file name on the `do_print` command line, the listing is sent to standard output.

A back end that produces C. Sometimes you may prefer to generate C rather than assembly language. This can be useful for developing a back end when the target is not available or for testing the output of one pass without first having to transform it through other passes. The `m2c` pass translates Machine-SUIF intermediate form into equivalent C code. For example, a good way to test the result of performing copy propagation at the `suifvm` level would be to use the following simple back end:

```
do_s2m $f.lsf $f.svm
do_dce $f.svm $f.scp
do_m2c $f.scp $f.scp.c
```

4.5 Constraints on pass orderings

Consistent with the SUIF philosophy, we have attempted to minimize restrictions on the ordering of Machine-SUIF passes. There are, however, certain constraints that must be met. For example, you must run `s2m` before any other Machine-SUIF pass. You must run `gen` before you run a register allocator, and you must run `fin` after both of these. Finally, you cannot run a base-SUIF pass on a Machine-SUIF IR file.

Again, please see the `src/machsuif/README.machine` file included with your distribution of Machine SUIF for information about the ordering constraints of the passes included in that distribution.

5 Developing a New Optimization Pass

One of Machine SUIF's key roles is to provide a programming interface, the OPI, for use in writing machine-level compiler optimizations. The OPI is not a compiler substrate like the SUIF compiler. It is designed to be implemented on top of a compiler substrate, and it relies on the substrate to take care of things like file I/O and the pipelining of passes. The OPI segregates the specification of optimization algorithms from the specifics of the infrastructure substrate that they happen to run on. Though you can directly manipulate the SUIF or Machine-SUIF IR, we recommend that you adhere to the OPI so that your efforts are usable by others.

The OPI includes data structures for implementing a low-level IR, e.g. instruction lists, instructions, operands, annotations, CFGs, etc. It defines functions for creating, inspecting, and manipulating these IR objects. The OPI relies heavily on plain old functions. This approach is efficient, it is easy to extend, and it leaves us able to revise major aspects of the implementation, should we need to, without disturbing optimization passes that already exist. The OPI also defines data structures for describing target-machine characteristics (e.g. the target machine's register file) and access functions for these data structures.

Often, the best way to create a new pass is to start with an existing one and reshape it. The `peep` pass distributed with Machine SUIF is a useful model. It illustrates how a pass is structured and how to use the control- and data-flow libraries. We also encourage you to look at *The Machine-SUIF Cookbook* [6].

Pass structure. A pass consists of two pieces: a substrate-independent piece that performs the actual analysis or optimization; and a substrate-specific wrapper that allows the substrate-independent piece to be called as a pass of the current compiler substrate.

We build each substrate-independent analysis or optimization pass as a stand-alone class. For example, the `peep` pass defines a class called `Peep` in the file `peep/peep.h`. This class, like our other substrate-independent pass classes, provides three basic methods: `do_opt_unit` which performs the actual peephole optimization on the specified optimization unit; `initialize` which allow us to write code that should run before calls to `do_opt_unit`; and `finalize` which contains code to be run after calls to `do_opt_unit`. Our current implementation of `Peep` uses `initialize` to clear some statistics-gathering variables that are printed in `finalize`.

The implementation of `peep` uses several Machine-SUIF libraries that contain valuable tools for optimization writers. For example, we provide libraries for building control-flow graphs (CFG), performing control-flow analysis (CFA), and gathering bit-vector-based data-flow analysis (BVD). Like the optimization passes, these libraries are parameterized (i.e., are target independent). You can find their documentation in the `src/machsuif/doc` directory. In particular, the `peep` pass uses the CFG and BVD libraries to collect liveness information. The CFA library is used when dominator analysis is needed, as in conversion to static-single-assignment form.

The code for the substrate-specific wrapper is contained in `suif_main.cpp`, `suif_pass.h`, and `suif_pass.cpp`. The file `suif_main.cpp` allows us to build the stand-alone program called `do_peep`. The `PeepSuifPass` class in `peep/suif_pass.h` wraps a `Peep` object in a subclass of SUIF's `PipelinablePass` class. This allows us to invoke `peep` as a pass in the SUIF compiler system. SUIF provides such classes as `PipelinablePass` to capture the boilerplate of processing command lines, reading and writing intermediate files, and sequencing through the procedures in each file. The methods of these classes are hooks on which to hang processing at strategic stages: before each file, before each procedure, and so on. The file `suif_pass.cpp` illustrates how we connect the sequencing methods in a SUIF `PipelinablePass` with those in `Peep`.

Binding the OPI and focusing the environment. To access facts about the target machine, each parameterized pass must create a `Context` object and install it into the global environment. Installation of the `Context` object binds the OPI to that particular target. We perform this binding in `suif_pass.cpp` during the processing of each SUIF `FileBlock`. Please note that this binding mechanism is particular to our current implementation of Machine SUIF; binding may occur in a different manner in other substrates.

We assume that all of the code within one input SUIF file corresponds to a single target, and we annotate each SUIF `FileBlock` with a Machine-SUIF `target_lib` annotation. This annotation specifies the target library with which to bind the OPI. If you look at the code in `do_file_block` in `suif_pass.cpp`, you see that we call the Machine-SUIF library function `focus(FileBlock*)` to inform the OPI implementation that we're now working within this `FileBlock`. This call also establishes the global `Context` object based on the target library string specified in the `target_lib` annotation on `the_file_block`. This code also appears below:

```
focus(the_file_block);
```

In addition to the focus call in `do_file_block`, we also make a focus call in `do_procedure_definition`. This call informs the OPI implementation of the specific procedure definition involved in the upcoming `do_opt_unit` call. Notice that in Machine SUIF's implementation of the OPI, an `OptUnit` is a `ProcedureDefinition`. We end `do_procedure_definition` with a call to `defocus`, which informs the OPI that the scope has moved back to the file level. The contents of `do_procedure_definition` in `suif_pass.cpp` are listed below:

```
focus(the_proc_definition);
peep.do_opt_unit(the_proc_definition);
defocus(the_proc_definition);
```

In summary, once you have indicated how the OPI should be "bound" and "focused", you are able to use functions like `is_move` and `target_regs`. They will work as expected for the current target. The `is_move` function provides a way for a parameterized copy-propagation algorithm, for example, to recognize register-move instructions in the target ISA. The `target_regs` object summarizes the hardware configuration of and software conventions for the current target's register set. For more information about the OPI, its setup, and its functions, please see *The OPI User's Guide* [8].

6 Extending the OPI

The OPI is comprised of two kinds of libraries: *interface* libraries and *implementation* libraries. The target libraries are all implementation libraries: those that implement the OPI functions for a single, particular machine target. Section 7 describes how to augment these kinds of libraries. In this section, we describe two ways in which you might extend the OPI by adding new functionality to the interface. Obviously, this functionality would be useful only if one or more target libraries define implementations for the new interface declarations.

Adding new functions. Suppose you wish to develop optimizations that take advantage of prefetch instructions. In such a case, you probably want to be able to ask if an instruction is a prefetch instruction or not. Analogous to the existing `is_move` function in the OPI, you create a new interface library (let's call it the `prefetch` library) that contains a declaration for a Boolean function `is_prefetch` that takes a single argument of type `Instr*`. Your optimization pass then simply links the new prefetch library to get access to this new `is_prefetch` function. It is straightforward to extend this example to add more sophisticated kinds of functions or objects. We encourage you to look at the machinery for existing functions such as `is_move` (an instruction predicate), `type_addr` (which yields the address data type for a target), or `target_regs` (which produces an object that describes a target machine's register files). We encourage you to develop analogous machinery for the functions and objects you want to support.

Notice that the inclusion of prefetch instructions in an architecture would probably cause you to want to change the implementation of the `reads_memory` OPI function, since prefetch instructions read memory, and the `reads_memory` predicate identifies such instructions. When developing the `prefetch` library however, you add only new functionality. You change the implementation of existing functionality, such as the `reads_memory` function, only in the target libraries, as explained in the next section.

Extending the IR The `prefetch` example requires adding a new opcode identifier and extending some target-specific internal tables (e.g., the table mapping an opcode to the corresponding ASCII string), but it does not require changing the underlying IR. You may wish however to optimize a target ISA that has an addressing mode not covered by those built into Machine SUIF. You could create a new kind of address-expression operand by defining a new subclass of the `Opnd` class and by providing methods (analogous to those for other such subclasses) that compose and decompose your new address expression, that distinguish them from other operands, and so on. In target libraries where your new operand kind is relevant, you need to accommodate it where target-specific treatment is appropriate. For example, you arrange to emit such operands in the ASCII form that the assembler expects to see. Again, note that you don't need to modify anything in the existing `machine` library to add a new kind of address expression.

You should read *The Extender's Guide* [5]. There you will find instructions for making common extensions such as the ones just mentioned.

7 Adding Support for New Targets

A new target architecture. You add support for a new target by creating a new target library. Here again, the best approach is probably to start from one of the existing libraries, such as the `alpha` library, and simply edit the implementations.

As mentioned above, a target library defines and creates a `Context` object, which provides the OPI functions with the means to act in a target-specific manner. The details of how we exactly accomplish this are interesting but not relevant to the current overview. It suffices to say that for each OPI function (e.g., `is_move`), you code a corresponding `Context`-member function.

For data structures that hold target-specific properties, such as opcode names and register-file descriptions, you write straightforward object-initialization code. These structures are declared in an interface library, and they are referenced through functions declared in those interface libraries. Whenever possible, you should use OPI functions in your code within the target library. In this way, others can benefit from your efforts as explained below.

The OPI documentation [8][5] and the `machine` library documentation [7] provide more detailed rationale for the contents of a target library.

A new variant of an existing target architecture. If you only want to specialize the description of an existing target by adding or changing characteristics of an existing implementation, then you should just supplement an existing target library. Machine SUIF relies on object-oriented techniques that make it easy to combine and refine the `Context` objects from existing libraries.

As an example, let's return to our previous discussion of adding support for prefetch instructions to Machine SUIF. Assume that your original `alpha` library does not contain prefetch instructions and that you wish to do the minimal amount of work necessary to create a target library called `alpha_pf`. This new library should be functionally identical to the `alpha` library except that it should also support Alpha architectures that contain prefetch instructions.

Within Machine SUIF, you are able to reuse the existing functionality from the `alpha` library that is appropriate for the `alpha_pf` library and override the functionality that needs to change (e.g., the implementation of the `reads_memory` function). You would also implement any new functionality unique to `alpha_pf` (e.g., the `is_prefetch` function).

Not only can you do this with minimal effort, but you also do this without having to alter the distributed Machine-SUIF libraries in any way. *The Extender's Guide* [5] contains further details and examples.

A new kind of machine instruction. To this point, our running prefetch example has only discussed how the system can support optimizations and analyses that are cognizant of prefetch instructions. Here, we say a few words about how the front-end might indicate to the back end that the back end should in fact instantiate and use a prefetch instruction at a particular point in the code.

Suppose that the pass that determines when and where to use a prefetch instruction is a base SUIF pass, and that you have already extended SUIF with a `PrefetchExpression`. You want this expression node to compile to the appropriate prefetch instruction for the targets of interest, and simply to be ignored for other targets. There is a `suifvm` instruction (with opcode `ANY`) for handling extensions of this kind. You would write an alternative lowering pass, either by replacing or by extending `s2m`, and you would have it translate each `PrefetchExpression` to a suitably-annotated `ANY` instruction in the `suifvm` dialect. The `instr_opcode` annotation of this generic `suifvm` instruction identifies it to target-specific code generators as a prefetch instruction; its operand sequences carry the necessary operands in the usual way.

8 More Radical Changes

Machine SUIF is a research compiler, and we don't claim to always know what's best. As such, we have made it possible for you to make fairly radical changes to the code base that we distribute. In particular, it is fairly easy to replace the implementation of our Machine-SUIF IR with a completely different implementation. It is also fairly easy to replace the underlying SUIF compiler environment with your own favorite compiler environment. The Deco project[7] at Harvard University in fact implements both of these changes.

Exactly what do we mean by "fairly easy"? We mean that you need only make changes to the code in the `machine` library; none of the code in the optimization passes nor the code in the target libraries (e.g., the `alpha` library) needs to change. The only pieces of the interface libraries (e.g., the `cfg` library) that would have to change are those pieces that extend the underlying IR. All of those are cleanly separated in each library.

As long as you adhere to the OPI, it insulates the bulk of the back-end passes and libraries from the structure of the actual IR and specifics of the underlying compiler environment. Generally speaking, we have parameterized all of our code with respect to the compiler environment; the "values" of these parameters are bound when you compile the optimization code base with a particular `machine` library. We have also parameterized our code with respect to the compilation target machine; the "values" of these parameters are bound when you use Machine SUIF to compile an application.

For further information, please refer to the Machine-SUIF documents describing the `machine` and `cfg` libraries.

9 Where To Look Next

In addition to the overview that you are reading, the `machsuif/doc` directory contains several other documents that the user of Machine SUIF might find helpful:

- *A User's Guide to the Optimization Programming Interface.* An introductory reference for the OPI. This document provides an in-depth look at the OPI from the perspective of someone who wants to use it to write optimizations.
- *The Machine-SUIF Cookbook.* A listing of several complete Machine-SUIF passes that will help you to begin writing your own optimizations.
- *An Extender's Guide to the Optimization Programming Interface.* A reference to the structure of the OPI. This document provides background material explaining how parameterization is accomplished, how the `Context` classes and objects are setup, and more detailed examples of how you can extend the OPI or enhance the system.

[7]See URL `http://www.eecs.harvard.edu/machsuif/research/deco.html`.

- *The SUIF Machine Library.* The definitive reference for the built-in OPI functions and data structures. This document describes the actual structure of the Machine-SUIF IR and the other key structures in Machine SUIF. It also describes classes that underlie some of the basic Machine-SUIF passes, including code finalization and code output (both in assembly language and in C). Finally, it describes a number of utilities that are provided by the `machine` library and widely used in the rest of the Machine-SUIF system.

- *The SUIFvm Library of Machine SUIF.* Describes the `suifvm` dialect, an idealized machine language not unlike the "low SUIF" dialect of SUIF version 1. This is the target language for lowering passes, and the source language for generators of real machine code. The class `CodeGen`, which is the foundation for the system's current concrete code-generation facilities is described here as well. This is the one library in our system that contains both extensions to the OPI (like an interface library) and an implementation of the OPI (like a target library).

- *The Machine-SUIF Control Flow Graph Library.* Documents the interfaces of the CFG library, which enables manipulation of programs through transformation of their flow graphs.

- *The Machine-SUIF Control Flow Analysis Library.* The CFA library is layered on the CFG library; it adds classes for natural-loop analysis and dominator analysis, including computation of dominance frontiers.

- *The Machine SUIF Bit-vector Data-flow-analysis Library.* The BVD library is also built on the CFG library. It provides a framework for classical iterative bit-vector-based data-flow analyses such as liveness, reaching definitions, and so on.

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] D. August et al. Integrated predicated and speculative execution in the IMPACT EPIC architecture. *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.

[2] M. Benitez and J. Davidson. Target-specific global code improvement: principles and applications. Technical report 94-42, University of Virginia, Charlottesville.

[3] P. Briggs and L. Torczon. An efficient representation of sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4), March-December 1993, pp. 59-70

[4] L. George and A. Appel. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3), May 1996, pp. 300-324.

[5] G. Holloway and M. D. Smith. *An Extender's Guide to the Optimization Programming Interface and Target Descriptions.* The Machine-SUIF documentation set, Harvard University, 2000.

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

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

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

[9] N. Ramsey. *Literate programming simplified.* IEEE Software, 11(5), September 1994, pp. 97-105.