Code Reuse in an Optimizing Compiler

Ali-Reza Adl-Tabatabai[1], Thomas Gross[1,2], and Guei-Yuan Lueh[3]

[1]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[2]Institut für Computer Systeme
ETH Zürich
CH 8092 Zürich

[3]ECE Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes how the `cmcc` compiler reuses code — both internally (reuse between different modules) and externally (reuse between versions for different target machines). The key to reuse are the application frameworks developed for global data-flow analysis, code generation, instruction scheduling, and register allocation.

The code produced by `cmcc` is as good as the code produced by the native compilers for the MIPS and SPARC, although significantly less resources have been spent on `cmcc` (overall, about 6 man years by 2.5 persons). `cmcc` is implemented in C++, which allowed for a compact expression of the frameworks as class hierarchies. The results support the claim that suitable frameworks facilitate reuse and thereby significantly improve developer effectiveness.

1 Introduction

A well-chosen set of application-specific frameworks results in significant code reuse, which is a requirement for concise code. In this paper, we report on our experience with using this software technology for an optimizing compiler. Optimizing compilers have a (well-deserved) reputation of being large and complex software systems. This complexity has three direct consequences: (1) maintenance of the compiler is expensive, (2) retargetting a high-quality compiler to a different instruction set architecture is difficult, and (3) experimentation with or extension of the compiler platform is often a challenge.

The need to address the first two items is obvious, but in production compilers, the third issue is important as well. Given the complexities of current processors, no single heuristic produces consistently superior results. Consequently, compilers try multiple heuristics and pick the best result. For example, the RS/6000 compiler [2] tries three different register-allocation heuristics, and then picks the allocation with the least spill code. The SGI MIPSpro compiler [29] uses 4 different scheduling strategies. Such an approach to structuring the compiler requires a high degree of flexibility (i.e., the ability to modify some part of the compiler while keeping the rest of the infrastructure unchanged). This flexibility is even more crucial in an aggressive *retargetable* compiler.

Various researchers and practitioners have embraced the paradigm of object-oriented design and implementation to achieve modularity and to increase code reuse [15, 28]. Generic class libraries for basic data structures such as linked lists, stacks, queues, heaps, and so on, provide a first level of code reuse [30, 26]. Such basic data structures are pervasive inside a compiler, and thus there is ample opportunity for code reuse based on libraries within a compiler. However, there exists a second level of code reuse based on *patterns* or *application frameworks* [12, 31, 32]. An application framework captures the control structure of a class of computations but leaves specification of the exact functionality to the client of the framework. For example, many optimization phases consist of one or more data-flow analysis passes followed by transformations based on the result of the analysis. Code can then be reused by developing a single application framework for global data flow. This framework factors out code that is common among different phases, such as iterations over the flow graph that modify the In and Out sets of each basic block. The framework is parameterized according to the characteristics of the flow analysis (e.g., the direction, confluence operation, etc.); a client of the framework then provides the details (e.g., in the form of a function to initialize the In and Out sets).

A. Adl-Tabatabai's current address: Intel Corporation, RN6-18, 2200 Mission College Blvd. Santa Clara, CA 95052-8119.

This research was sponsored in part by the Advanced Research Projects Agency/ITO monitored by SPAWAR under contract N00039-93-C-0152.

Our goal for the cmcc compiler was to develop a competitive optimizing C compiler that is useful as a platform for architecture and compiler research. This goal implies the need to support easy retargetting (to generate code for different machines). Given the small number of people involved (the authors), we strive for high internal (between modules) and external (between versions for different targets) code reuse. Our definition of code reuse is rather strict — we do not permit any modification to reused code, since the compiler supports multiple targets. If the reused code is included unchanged in all compilers, then all versions of the compiler can benefit from advances or changes. This situation is different from those environments where "stand-alone" applications are built: Two systems may start with the same class library but then are permitted to change it according to their respective needs.

We note that there is no generally accepted definition of reuse. Even changes of up to 25% of the source code have been classified in the literature as reuse, with "slight revisions" [23]. Although such a form of reuse is better than developing code from scratch, it nevertheless violates our requirement that reuse is based on complete, unchanged use of the code base. In this paper, we report how application frameworks increase code reuse and ease compiler extensions (i.e., support the addition of new optimizations, different orderings among transformations, and new targets). Since we do not allow any modification to the reused code, our experience reported here makes a conservative statement about the reuse through framework technology. Other environments might report even higher degrees of reuse if slight modifications are allowed.

In the next section, we briefly establish that cmcc is a competitive optimizing compiler. Then, in the subsequent sections, we describe the use of frameworks in the global data-flow analysis, code generation, instruction scheduling, and register-allocation modules of the compiler.

2 The cmcc compiler

We developed an optimizing, retargetable compiler (the Carnegie Mellon C Compiler or cmcc) for a complete programming language (ANSI C), with all the warts and subtleties that a real language implies. Results for language subsets (e.g., pointer-less languages) are difficult to extrapolate to other compilers. cmcc is coded in C++ and produces assembly language for a variety of target machine architectures. It uses the lcc ANSI C front end [10]. Table 1 lists the global optimizations performed by cmcc.

The global register allocator integrates live range splitting [24] with a graph-coloring register allocator [6]. Register allocation is done as part of code generation (although it is listed here as an optimization, it is not part of the global optimizer).

Loop unrolling and peeling
Linear function test replacement
Induction variable expansion
Induction variable simplification
Constant propagation and folding
Induction variable elimination
Assignment propagation
Partial dead code elimination
Dead assignment elimination
Partial redundancy elimination
Strength reduction
Branch optimizations
Instruction scheduling
Global register allocation (using graph coloring)
Register coalescing

Table 1: Optimizations performed by cmcc

	MIPS		SUN
program	gcc -O2	cc -O	cc -O
alvinn	1.07	0.94	0.79
compress	0.84	0.95	1.04
ear	1.08	0.95	0.99
eqntott	1.14	1.09	0.59
espresso	1.06	1.05	1.02
li	0.98	1.05	1.16
sc	1.07	1.03	1.00

Table 2: Performance of optimized code generated by cmcc, relative to optimized code generated by gcc (version 2.3.2) and MIPS cc on a DECstation 5000/200, and SUN cc on a SPARC.

Table 2 contrasts the optimized code generated by cmcc with the optimized code produced by gcc and the native MIPS cc compiler for the MIPS architecture, and with the code produced by the native SUN cc compiler for the SPARC architecture. (This table presents the performance relative to these compilers; a number of less than one means that cmcc produces better code.) The sample programs are the C programs of the SPEC suite. Overall, even without serious performance tuning, cmcc produces code that is roughly of the same quality as the code produced by the other compilers.

The global optimizer's interface to cmcc's code generator is encapsulated inside an abstract code generation class (this is an example of an *interface class* [31]). Each target's code generator inherits from this class. The register allocator and instruction scheduler are machine independent and operate on schedules of abstract instructions. To test our claims of retargetability, we chose two extremes of the range of possible target machines. cmcc currently generates code for various RISC machines (MIPS, SPARC, DLX) and an experimental LIW machine (Intel iWarp).

3 The data-flow analysis framework

The majority of global optimizations listed in Table 1 are based on bit-vector data-flow analyses. Such optimizations are similar in structure, and we can reuse code by developing an application framework (the data-flow analysis (DFA) application framework) for such problems.

3.1 Description

Informally, a DFA problem is defined by several parameters (see the appendix for a more detailed description):

Direction The direction of a DFA can be either forward or backward. Examples of forward analyses include available expressions and reaching definitions. Example of backward analyses include anticipated expressions and live variables.

Confluence The confluence operation can be either the conjunction or disjunction operator, resulting in either an all paths or any paths problem respectively. Available expressions and busy expressions are all paths problems, while reaching definitions and live variables are any paths problems.

Initial In and Out sets The initial value of the In (Out) bit vector at the source (sink) node in the control-flow graph depends on the data-flow problem. For instance in live variables, global variables are considered live at the sink node and local variables are not live, while in available expressions all expressions are considered not available at the source node. The initial values of the In and Out bit vectors at all other nodes depend on the confluence operator.

Gen and $Kill$ sets The Gen and $Kill$ sets are defined by the data-flow analysis problem.

Transfer functions The transfer functions relate the In and Out sets at each node in the control-flow graph. The most common transfer functions are $Out = (In - Kill) + Gen$ for forward flow problems and $In = (Out - Kill) + Gen$ for backward flow problems.

To solve a DFA problem, the transfer functions are repeatedly solved at each node in the control-flow graph until a fixed point is reached. To speed up this computation for forward flow problems, the transfer functions can be solved in reverse post order of the control-flow graph. To speed up the solution for backward flow problems, the blocks can be visited in the reverse order of the forward flow case.

In general, unidirectional DFA problems can be categorized into four types according to their direction and confluence operator: (1) forward flow all paths, (2) forward flow any path, (3) backward flow all paths and (4) backward flow any path. DFA-based optimizations differ mainly in their Gen and $Kill$ sets, the initial value of the In (Out) set at the source (sink) node in the control-flow graph, and how the results of the data-flow analysis are used.

`cmcc`'s DFA framework is organized as a C++ class hierarchy, as depicted in Figure 1. A new DFA problem is defined as a C++ class that inherits from one of four classes: `Fwd_And_Data_Flow`, `Fwd_Or_Data_Flow`, `Bwd_And_Data_Flow`, and `Bwd_Or_Data_Flow`. Each one of these classes corresponds to one of the DFA problem categories defined above. These classes, however, are *abstract classes*; that is, they contain declarations of virtual functions whose implementations must be supplied by inheriting classes. To create a concrete class, an inheriting class must specify a virtual function for initializing the Gen and $Kill$ sets at each block, a virtual function for allocating bit vectors, and a virtual function for using the results of the DFA at each basic block.

To specify most DFA problems, one need only to define the three virtual functions described above. However, not all DFA problems are so neatly similar, and some need to specify other parameters. Some problems need more complicated transfer functions; for instance, lazy code motion as described in [17], or bidirectional data-flow problems such as the classic partial redundancy algorithm of [25].

Other problems need to define the initial value of the In (Out) set of the source (sink) node (e.g., live variables). To satisfy these types of problems, an inheriting class can override the default virtual function for performing the transfer function at a block, and the default virtual function for initializing the In or Out set of the source or sink node (depending on the direction of the DFA).

Figure 2 highlights those member functions and fields that are of interest for this class. The `GEN`, `KILL`, `IN` and `OUT` arrays hold pointers to bit vectors for each basic block and are indexed by basic block numbers. Bit vectors are allocated by the `init()` method. `fg_iter` is a generic iterator that iterates over basic blocks. This member is initialized by inheriting classes, to an iterator that iterates in the forward or backward direction in the control-flow graph. The `confluence()` and `transfer()` methods perform the confluence operation and transfer functions at a basic block. `trans()` is a default transfer function that can be used by inheriting classes.

The fixed point computation is performed by the `dfa()` method; the definition of this method is shown in Figure 10 in the appendix. When the fixed point is computed, `propagate()` is called for each basic block. An inheriting class uses the results of the data-flow analysis in the `propagate()` method.

In the next level of the hierarchy, there are the `Fwd_Data_Flow` and `Bwd_Data_Flow` classes, which inherit from the `Data_Flow` class. In the appendix, Figure 11 shows `Fwd_Data_Flow`, the other class is symmetric.

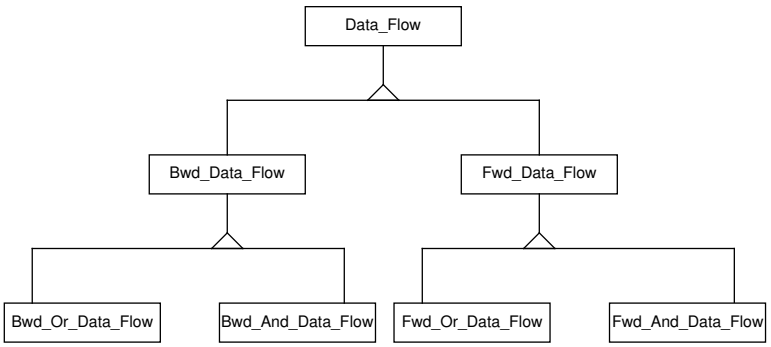

Figure 1: Overview of class hierarchy.

```
class Data_Flow {
  protected:
    BV  **GEN,**KILL,**IN,**OUT,*tmp;
    Block *src_block, *sink_block;
    Block_Iter *fg_iter; // iterates over blocks in CFG
    virtual void confluence(Block *b) = 0;
    virtual int  transfer(int block_id) = 0;
    BV *trans(BV *in,BV *gen,BV *kill)
        {*tmp = *in;*tmp -= *kill;*tmp += *gen;return tmp;}
    virtual void init_Root() = 0;
    virtual void init_In_Out(BV *in,BV *out) = 0;
    virtual void init_Gen_Kill(Block *b,BV *gen,BV *kill) = 0;
    virtual int  propagate(Block *) = 0;
    virtual BV*  new_BV() = 0;
  public:
    Data_Flow(Block_Iter *fg_iter);
    void dfa(); // computes the fixed point
    void init(); // allocate bit vectors
};
```

Figure 2: The `Data_Flow` base class.

These two classes specialize the `Data_Flow` base class by defining the direction of the DFA, and thus define all the virtual functions in the `Data_Flow` class that depend on the direction of the DFA: `confluence()`, `init_Root()` and `transfer()`. (Note that `transfer()` is kept as a `protected` member so that inheriting classes can override this method if necessary.) Since the confluence operation is not yet known, a new virtual function `confl_Op()` is defined.

The third level of the hierarchy adds the confluence operation to the direction, resulting in four classes ($\{\texttt{Fwd}, \texttt{Bwd}\} \times \{\texttt{Or}, \texttt{And}\}$). Figure 3 provides an example of how the DFA framework is applied to the available expressions problem. The few lines of code shown in this figure represent the core of the optimization.

3.2 Reuse

Table 3 shows the size (in lines of C++ code) of the DFA framework. These numbers do not include the common standard templates (e.g., lists, iterators, etc.), since we concentrate on the importance of frameworks in this paper. Lines of code are a simple metric to assess reuse. If the frameworks are reused by a different team, other metrics like development time, rate and number of ECO (engineering change orders) would provide more insight in the reusability of the system.

There are two noteworthy aspects: First, the code is extremely compact (and also highly efficient). Second, most of the code is placed in the base class, reflecting several iterations of refinement and (re)factoring. As alluded to in Figure 3, the use of the DFA framework also allows for compact code in the optimizer, since the framework forms the foundation

Level	Lines of Code
Base	230
Fwd/Bwd direction	111
3rd level	28
Total	369

Table 3: Size of DFA framework.

for a wide range of optimizations. Table 4 lists the sizes of the parts of the optimizer that use the DFA framework.

Optimization	# of DFA steps	Lines of Code
Assignment propagation	1	430
Dead assignment elimination	1	327
Assignment sinking	2	570
Partial redundancy elimination	4	2506
Live range analysis	2	331
Debugger	5	1836

Table 4: Size of clients of the DFA framework.

Overall, there are 10 global data-flow analysis passes in `cmcc`. In addition, 5 more data-flow analysis passes are done to implement algorithms for detecting the effects of optimizations on source-level debugging [1]. Comparing the code for the various optimizations, and the number of DFA steps for each optimization, we see that `cmcc` exhibits significant reuse. Another important aspect is not visible in the plain numbers: since DFA is so compact and efficient, we never thought twice about using DFA. As a result, the optimizer is concise (and we even found new uses for DFA: enabling source-level debugging of optimized code).

4 Code generation framework

There are two challenges for the code generator: modeling the target machine and describing a way to select an appropriate code sequence for each intermediate representation (IR) node. Moreover, since we would like our compiler to be retargetable, we would also like the code generator to address these issues in a machine-independent manner.

4.1 Description

The machine-dependent aspects of the code generator are represented by a set of *abstract* classes that hide and encapsulate target machine details. Code selection, register allocation, instruction scheduling, and code emission then become totally machine independent by manipulating these abstract classes. Machine-dependent aspects include machine instructions, register banks, instruction selection, and so on. We highlight here the two principal abstract classes in the code generator: `Inst` and `Code_Generator`; all target machines must inherit from these two abstract classes.

The `Inst` class hierarchy

The abstract `Inst` class is the code generator's machine-independent representation of an instruction. The machine-independent functions inside the code generator operate on trees and schedules of machine-independent instructions of type `Inst`. The `Inst` class encapsulates all the information that is needed by the machine-independent phases of the code generator. Its interface is a set of methods that allow clients to query instruction characteristics such as the size of immediate(s), the register bank of result register, the commutability, the result latency, and so on. The `Inst` class also includes methods for code emission:

```
class Inst {
    Inst    *src[MAX_SRCS];
    unsigned n_srcs;
    Register *dst_reg;
    . . .
  public:
    virtual void emit(ostream&) = 0;
    virtual Register_Bank *bank() = 0;
    virtual int is_commutable() = 0;
    virtual int imm_n_bits() = 0;
    virtual int imm_sign_extend() = 0;
    . . .
};
```

Each target machine inherits from this base class and provides the definitions of this class' pure virtual functions; Figure 4 shows the class hierarchy. Similar instructions are grouped into a shared class, simplifying the hierarchy (e.g., `Mips_Convert_Inst` contains all the data-format conversion instructions of the MIPS architecture). A class `X_Special` captures those that do not fit; note that there is no class `MIPS_Special`, all the MIPS instructions fit neatly into one of the other classes. The MCode intermediate representation, which was developed independently as part of the Clarity C++ project, converged to an instruction set that closely resembles this group of instructions [22].

The constructor for each of these classes generates an instruction (node) of the respective type. The data element `_Info` keeps information about this class of instructions, such as the assembly opcodes, whether an instruction accepts immediate values, which register bank (if any) holds the operands, format strings for the code emitter, instruction latency, and so on. This allows a class higher up in the hierarchy to implement most of the virtual functions of the `Inst` class (see Figure 13 in the appendix for an example).

The register allocator, scheduler, and emitter operate on objects of type `Inst`. Since the machine-specific instruction

```
class Avail_Exprs : public Fwd_And_Data_Flow {
  public:
    Avail_Exprs(Block *src_blk) : Fwd_And_Data_Flow(src_blk) {}
    int  propagate(Block*) {}
    void init_Gen_Kill(Block *block,BV *gen,BV *kill);
    BV   *new_BV() {return new_Expr_BV();}
} ;
```

Figure 3: An example application of the DFA framework: Available expressions.

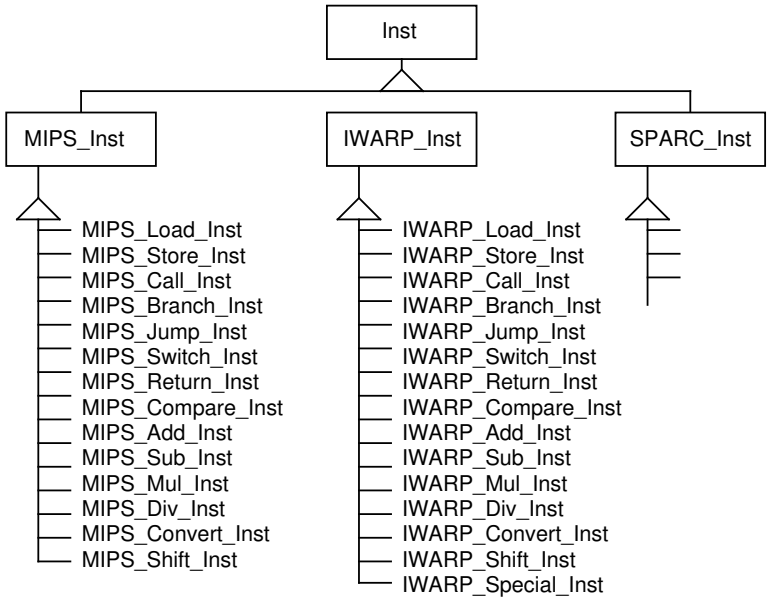

Figure 4: Hierarchy of machine instruction classes.

classes are all derived from `Inst`, this choice allows us to keep these other modules machine independent.

The `Code_Generator` class hierarchy

The abstract `Code_Generator` class encapsulates all functions that generate machine instructions; it includes methods to perform arithmetic operations, to copy a block of memory, to get argument registers, and so on. All code is generated through this code generation class: whenever a code generator function needs to generate code, it uses methods of the class `Code_Generator`. For example, the register allocator uses this class to generate spill, caller-save, and callee-save code in a machine-independent manner. Figure 5 shows a fragment of the `Code_Generator` class.

A benefit of using an abstract code generator class is that the compiler back end can dynamically select the target code generator, providing the mechanisms to implement a *fat compiler* (Section 7). That is, the compiler object can contain multiple code generators at the same time. The template `Code_Generator_For_` shown in Figure 6 allows to conveniently declare machine-specific code generators.

As an example of how these abstract classes are used we consider the code selector. Code selection is driven by the `match()` function, which matches the nodes of the IR with instructions of the target machine. IR nodes are defined by the class `Node`, and the set of operators of the IR nodes are the same as the ones defined by the `lcc` compiler [10]. The function `match()` takes an IR node and abstract code generator, and returns abstract instructions generated for the IR node:

```
Inst *match(Node *n,Code_Generator& cg);
```

The code selection function `match()` includes the following code for selecting floating point conversion:

```
return cg.cvtdf(match(n->left,cg));
    // returns instruction that
    // converts left subtree
```

```
struct Code_Generator {
    const MachineKind machine;
    Code_Generator(MachineKind mc) : machine(mc) {}

    ...   // many omitted
    virtual Inst *or(Inst*,Inst*) = 0;          // bitwise inclusive or
    virtual Inst *ld8(int is_signed,Inst*) = 0; // load byte
    virtual Inst *cvtdf(Inst*) = 0;             // convert double to float
    virtual Inst *cvtdi(Inst*) = 0;             // convert double to integer
    ...
}
```

Figure 5: Fragment of `Code_Generator` class.

Each machine-specific code generator realizes the virtual functions defined in the abstract `Code_Generator` class. For example, the MIPS and iWarp code generators contain:

```
Inst
*MIPS_Code_Generator::cvtdf(Inst *src){
    return new MIPS_Convert_Inst
      (MIPS_Convert_Inst::cvtds,src);
}

Inst
*IWARP_Code_Generator::cvtdf(Inst *src){
    return new IWARP_Convert_Inst
    (IWARP_Convert_Inst::cvtd2f,src);
}
```

`cvtds` and `cvtd2f` are names chosen by the implementors of each code generator respectively; these names are usually chosen based on the mnemonics of the assembler. In the above case, the two methods look almost identical, because both machines support a direct conversion from double to float. Conversions between from double to integer (`cvtdi`) are different. In this case, on the MIPS the (integer) result ends up in a special register and must be moved to an integer register; on the iWarp, there is only a single register bank:

```
Inst
*MIPS_Code_Generator::cvtdi(Inst *src){
    src = new MIPS_Convert_Inst
     (MIPS_Convert_Inst::cvtdw,src);
    return new MIPS_Mov_Inst
        (MIPS_Mov_Inst::mffp_lo,src);
}

Inst
*IWARP_Code_Generator::cvtdi(Inst *src){
    return new IWARP_Convert_Inst
     (IWARP_Convert_Inst::cvtd2w,src);
}
```

4.2 Reuse

Tables 5 and 6 present the number of lines of code for the machine-specific parts and the common part of the code generator.

Target	Lines of Code
MIPS	3072
iWarp	3256
SPARC	3578

Table 5: Size of code generators.

The category "Misc" in Table 6 includes the generic base classes for instructions (`Inst`), registers (`Register`), register banks (`Register_Bank`), and so on. For each code generator, approximately 75% of the code is reused. The small number of lines for each code generator renders the use of code-generator generators unnecessary.

Module	Lines of Code
Code emitter	575
Code selector	1165
Scheduler	970
Register allocator	7640
Misc	4245
Total	14595

Table 6: Size of generic code generator.

Another data point to gauge the difficulty of producing a code generator is that one of the authors developed the SPARC code generator in about a week (after developing the MIPS code generator). The other code generators evolved as part of the compiler development, so their development time is not a meaningful metric.

```
template <MachineKind MACHINE>
struct Code_Generator_For_ : Code_Generator {
    Code_Generator_For_() : Code_Generator(MACHINE) {}

    ...
    Inst *or(Inst*,Inst*);
    Inst *ld8(int signed,Inst*);
    Inst *cvtdf(Inst*);
    ...
}
// Mips code generator declarations

typedef Code_Generator_For_<MIPS> MIPS_Code_Generator;
MIPS_Code_Generator _mips_code_generator;

// Iwarp code generator declarations

typedef Code_Generator_For_<IWARP> IWARP_Code_Generator;
IWARP_Code_Generator _iwarp_code_generator;

// Functions for selecting code generators

Code_Generator& mips_cg()  {return_mips_code_generator;}
Code_Generator& iwarp_cg() {return _iwarp_code_generator;}
```

Figure 6: Declarations of code generators.

5 Scheduling framework

Instruction scheduling is an important optimization for pipelined and superscalar machines. Schedulers, however, are often handcrafted and fine-tuned, so this part of a compiler deserves attention by the developers of application-specific frameworks.

In the `cmcc` compiler, code scheduling is a separate code generator phase that takes and produces schedules of instructions. Like the other code generation phases, scheduling operates on the `Inst` class, and is thus machine independent. The purpose of scheduling is to reorder instructions such that the target machines' resources are used more effectively. Therefore, the scheduler must accurately model the result latencies and resource requirements of each instruction, while still being machine independent. Moreover, for multiple-issue (i.e., superscalar and VLIW) machines, the scheduler must accurately model the instruction fetch and decode constraints of the target machine. For a VLIW machine, the scheduler packs instructions (since each instruction contains multiple operations), while for a superscalar machine, the scheduler schedules instructions into an instruction window that models the instructions that are fetched and decoded as a unit by the processor at runtime (the size of the window is determined by the implementation of the machine).

5.1 Description

Scheduling must preserve two kinds of constraints: precedence constraints and resource constraints. Precedence constraints indicate the minimum latency between an instruction that produces a value and instructions that use the value. We model the precedence constraint between two instructions with an arc that is annotated with the result latency (in clock cycles) of the producer instruction. Constructing precedence arcs is done in a machine-independent manner; the `Inst` abstract class (see previous section) encapsulates all the necessary dependence and latency information.

Resource constraints ensure that there are enough machine resources (e.g., decode slots, functional units, result buses, etc.) for each scheduled instruction. Resources are machine dependent, so `cmcc` models machine resources with another abstract interface class called the `Resource` class. This class declares two virtual functions that the scheduler uses for manipulating resources: `struct_hazard(Resource*)` and `subscribe(Resource*)`. The `struct_hazard()` function takes a request for resources (at a particular slot in

the schedule) and checks whether the resources are available. The `subscribe()` function books the request. Each target machine inherits from the `Resource` class and defines the implementation of these two virtual functions. The `Inst` and `Resource` abstract classes allow the code scheduler to perform scheduling in a machine independent fashion.

List scheduling is the basic local scheduling technique used in `cmcc` (and many other compilers); it consists of four steps: (1) initializing a ready list that holds all instructions that are ready to be scheduled (ready instructions), (2) choosing a ready instruction I based on a *candidate selection heuristic*, (3) inserting I into the schedule after making sure the resource constraints of I are satisfied, and (4) adding into the ready list those instructions whose precedence constraints are satisfied by the scheduling of I.

The *direction* of code scheduling is determined by the order in which instructions are considered (i.e., placed on the ready list). If instructions are placed on the ready list after their descendants are scheduled, the code scheduler is a *forward* scheduler. In this case, the scheduler starts with the producers of values. If instructions are place on the ready list after their parents are scheduled, the scheduler works *backward*; that is, it starts with the consumers of values.

The manner in which candidate instructions are selected and inserted into the schedule in steps 2 and 3 of the list scheduling algorithm can be either *instruction oriented* or *cycle oriented*. An instruction-oriented scheduler chooses a ready instruction (using the candidate selection heuristic), inserts the instruction into a schedule slot such that the precedence and resource constraints are satisfied, and then modifies the ready list. A cycle-oriented scheduler steps through the schedule cycle-by-cycle. For each cycle, it chooses a ready instruction whose precedence and resource constraints are satisfied in the current cycle (using a candidate selection heuristic to break ties if there is more than one such instruction), inserts the instruction into the schedule, and then modifies the ready list.

The `Scheduler` class hierarchy is depicted in Figure 7. The `Scheduler_FWD` and `Scheduler_BWD` classes specialize the scheduler based on the direction (forward or backward). The bottom level of the hierarchy determines whether the scheduler is cycle or instruction oriented.

The `Scheduler` class declares two pure virtual functions that determine the candidate selection heuristic but are not defined in the class hierarchy: `choose_cand()` and `init_cand_heuristic()`. Clients define the candidate selection heuristic by defining these two functions. Among the possible candidate selection heuristics are maximum delay from root, number of precedent instructions, and result latency.

Building one or more schedulers using this framework is straightforward: simply inherit from one of the four bottom levels of the hierarchy and provide the candidate selection heuristic. In this manner, a compiler can "mix-and-match" from a number of different scheduling strategies and heuristics, and try different approaches for either different target machines (if they have different hardware features) or even different programs [29].

Note that this scheduling framework almost completely covers the range of possible local list schedulers.

5.2 Reuse

Table 7 shows the size of the scheduler framework. The category "Misc" includes bookkeeping, computation of precedence arcs, and so on. Given the framework, specifying a scheduler requires less than 100 lines of code, as can be seen in Table 8, which lists the sizes of several schedulers built as part of the project.

Level	Lines of Code
Base	282
Fwd/Bwd direction	60
Cycle/Inst. orientation	31
Misc	597
Total	970

Table 7: Size of scheduler framework.

Scheduler heuristic	Kind	Lines of Code
Number of descendants	Fwd_Inst	56
Execution time	Fwd_Cyc	55
Max path from root	Bwd_Inst	66
Max bwd. operation latency	Bwd_Inst	68
Max bwd. cycle latency	Bwd_Cyc	69

Table 8: Sizes of different schedulers.

The amount of code reuse depends also on the aggressiveness of the compiler and the tolerance of the user for compilation speed. Clearly, as a compiler attempts to find the best schedule based on a number of heuristics [29], this framework is reused multiple times in a single compiler. `cmcc` currently performs only one scheduling step, but we have implemented several scheduling heuristics and are actively investigating the application of multiple scheduling heuristics to a single compilation unit.

6 Register-allocation framework

Register allocation is an important optimization for modern processors that exhibit high memory access latencies. Like instruction schedulers, register allocators are usually hand-crafted and extensively fine-tuned. In the `cmcc` compiler, register allocation is performed after code scheduling. The `cmcc` register allocator takes a scheduled code sequence in which instructions use virtual registers and assigns physical registers based on graph coloring.

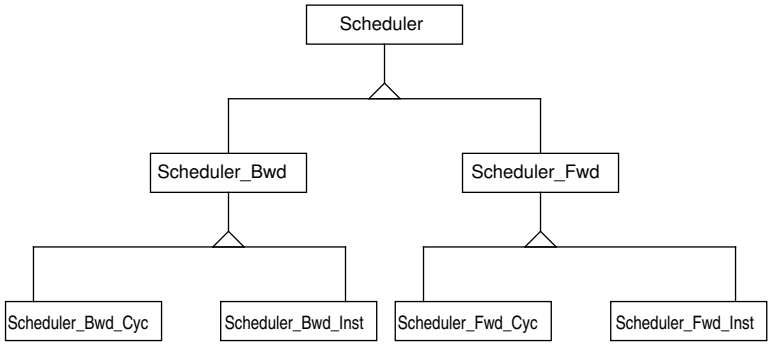

Figure 7: Hierarchy of scheduler classes.

There exist a number of approaches to graph-coloring based register allocation; for example, Chaitin-style coloring [7, 2], optimistic coloring [4], priority-based coloring [9] and several modern approaches [3, 5, 27, 24] that take the program structure into account when splitting live ranges. It is difficult and impractical to implement and experiment with these different register-allocation approaches without a register-allocation framework. In addition, a framework facilitates a *fair* comparison; that is, if an improvement is added, we would like *all* register allocators to benefit from the improvement rather than biasing in favor of a single approach.

6.1 Description

We identify seven phases in the register allocator: *graph construction*, *live-range coalescing*, *color ordering*, *color assignment*, *graph reconstruction*, *spill-code insertion*, and *shuffle-code insertion* (as illustrated by Figure 8). The graph-construction phase builds the interference graph for the input instruction sequence. The color-ordering phase heuristically determines the order in which live ranges are to be assigned colors. The color-assignment phase assigns colors to live ranges based on the ordering computed by the previous phase; this phase uses the interference graph to make sure that conflicting live ranges are assigned different colors. If either of the color-ordering or color-assignment phases spills a live range, the register allocator rebuilds the interference graph and restarts from the color-ordering phase. Our register allocator reserves no registers for spill code; that is, the register space is not divided into "local" and "global"registers.

There are two ways to reconstruct the interference graph: (1) spill code (a load/store) is inserted into the schedule immediately before/after a use/definition, the interference graph is thrown away, and the whole coloring process restarts from the first phase; (2) the existing interference graph is modified by producing a new small live range for each occurrence of a spilled live range (no spill code is inserted; the newly produced live ranges span only one instruction), and coloring proceeds with the modified graph. The first approach reuses the code for constructing the interference graph. The second approach favors compilation speed, because the interference graph is not rebuilt from scratch. `cmcc` takes the second approach (i.e., it reuses the graph, to save compilation time).

Spill code and shuffle code, respectively, are inserted into the final schedule by the last two phases, spill-code insertion and shuffle-code insertion. Shuffle code moves a data value between the different storage locations assigned to a split live range.

There are two stacks that are used as the interface between the color-ordering and color-assignment phases: (1) the `color` stack (C), and (2) the `spill` stack (S). The color-ordering phase determines the order in which live ranges are assigned colors and pushes live ranges onto the C stack based on the ordering — the higher the position within C, the higher the priority. The color-assignment phase pops live ranges from C and finds legal colors (i.e., non-conflicting colors) for them. The S stack is a pool for keeping spilled live ranges. If the color-ordering phase decides to spill live ranges, or the color-assignment phase fails to find legal colors for live ranges, those spilled live ranges are pushed onto S.

6.2 Modeling existing register-allocation approaches

This section elucidates how some existing register-allocation approaches fit into the register-allocation framework. A comparative evaluation of the various approaches is beyond the scope of this paper.

Chaitin-style coloring [7, 2] uses a *simplification* process to determine the coloring order. Simplification removes all *unconstrained* live ranges (degree $< N$); when a node is removed, all edges incident upon the node are also removed, opening up further opportunities for simplification. If simplification blocks (all nodes have degree $\geq N$), a least-cost live range is spilled so that the simplification process can proceed. This approach makes spilling decisions in the color-ordering phase.

Optimistic coloring [4] also uses the simplification process

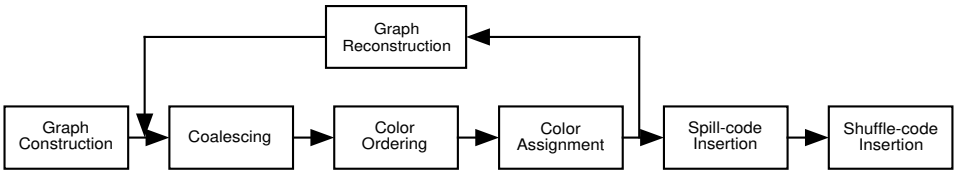

Figure 8: Structure of the register allocator.

to determine the coloring order. When the simplification process blocks, however, rather than spilling, optimistic coloring pushes a least-cost live range onto the C stack, removing the live range from the residual graph. In this manner, a live range that would have been spilled by the Chaitin-style approach is given another chance at receiving a register in the color-assignment phase. Optimistic coloring defers spill decisions until the color-assignment phase, which spills those live ranges that cannot be assigned legal colors.

Priority-based coloring [9] determines the coloring order according to a priority function. This approach first seperates out the unconstrained live ranges since unconstrained live ranges can always be assigned a legal color. If a live range cannot be assiged a legal color (because of conflicts with the neighbors' colors), then priority-based coloring splits this live range.

The framework shown in Figure 9 is general enough to model the preceeding approaches to coloring-based register allocation[1]. This framework also underlines the similarity of priority-based coloring to optimistic coloring; the difference is that one color ordering is simplification based, whereas the other is priority based. In addition, this framework can also model other approaches to live-range splitting that take the program structure into account, such as the approaches taken by Tera [5], Multiflow [11], and RAG [27], as well as splitting based on the SSA representation [3], and the integrated approach to splitting and spilling provided by fusion-style coloring [24]. A more detailed discussion of these approaches can be found in a technical report [24].

The class hierarchy of the register-allocation framework is depicted in Figure 9. There are two classes: (1) `Graph_Coloring`, which provides the fundamental framework for coloring-based register allocation, and (2) `Graph_Builder`, which is responsible for building the interference graph. The `Graph_Coloring` class (Figure 14 in the appendix) is the engine that drives the register allocation. All phases are implemented in `Graph_Coloring` except the color-ordering and shuffle-code insertion phases; these two phases are declared as virtual functions: `color_ordering()` defines the method for determining the color ordering, and `shuffle_code_insertion()` defines the method for inserting shuffle code for the split live ranges. Each of the various approaches to register allocation is encapsulated in a class. The class `Priority_Based` inherits from `Graph_Coloring` and provides an implementation of `color_ordering()` that computes the color ordering using a priority function. `Simplification_Based` also inherits from `Graph_Coloring` and provides a version of `color_ordering()` that uses the simplification process to determine the color ordering. The `shuffle_code_insertion()` methods of the `Priority_Based` and `Simplification_Based` classes are *nop* functions because no live ranges are split by these approaches to register allocation. The class `Fusion_Based` [24] inherits from `Simplification_Based` and adds an implementation of `shuffle_code_insertion()`.

The register-allocation framework allows live-range splitting in the graph-construction phase; we view live range splitting as a way to build an interference graph that has fewer interferences, or potentially requires fewer overhead operations. The class `Graph_Builder` has two functions, `construction()` and `reconstruction()`, which correspond to the graph construction and reconstruction phases, respectively. `construction()` is declared as a virtual function. The class `Conventional` inherits from `Graph_Builder` and constructs the conventional interference graph, which does not have split live ranges; priority-based, Chaitin-style, and optimistic coloring use this kind of graph construction. The program structure based approaches to live range splitting require different mechanisms for building the interference graph; for example, RAG needs the program dependence graph (PDG) structure [27], the Tera approach requires the loop hierarchy [5], and the Multiflow register allocator needs a trace picker [11]. The class `Splitting_Style` provides the framework for splitting live ranges during construction of the interference graph. The classes `Single_Bb`, `Loop`, `Ssa`, `Pdg`, and `Trace` all inherit from `Splitting_Style` and provide the appropriate method to construct the interference graph in the desired form.

6.3 Reuse

Table 9 lists the sizes of different coloring approaches. The "Simplification Based" and "Priority Based" entries give the size of the code that invokes the simplification process and

[1] Priority coloring as modeled here differs in one aspect from the description in [9]: it excludes live range splitting (i.e., live ranges are spilled instead of split when the supply of colors is exhausted).

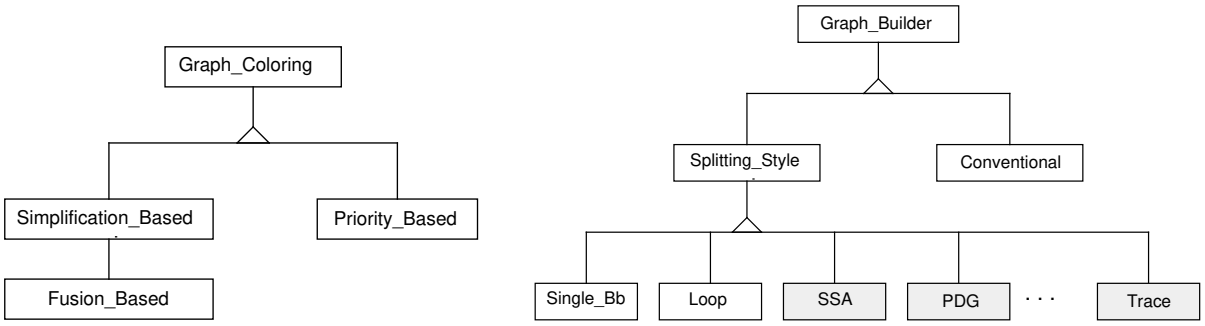

Figure 9: Class hierarchy of register-allocation framework.

sorting routine, respectively. The "Fusion Based" category contains primarily the code for shuffle-code placement. The "Misc" category consists of classes for live ranges, interference edges, assigned-color information, etc.

Graph Coloring	Lines of Code
Base	2543
Simplification Based	40
Priority Based	71
Fusion Based	992
Misc	1859

Table 9: Size of `Graph_Coloring`.

Table 10 gives the size of the hierarchy of graph builders. Presently, the `cmcc` compiler implements the "Single_Bb" and the "Loop" approaches to live-range splitting (i.e., single basic blocks or loops define the extent of a live range unit; see [24] for more details).

Graph builder	Lines of Code
Base	1281
Splitting Style	471
Conventional	18
Single Bb	19
Loop	39

Table 10: Size of `Graph_Builder`.

Table 11 shows the size of each phase of register allocation. The two entries for "Color ordering and "Shuffle-code insertion" are zero because these virtual functions must be defined by each client (register allocator). Some phases that are not shown in Figure 9 are classified into the "Misc" category (e.g., insertion of caller-save code, data-flow analysis for building the interference graph, static estimation of execution frequency (if no profiling information is available), selecting different register-allocation approaches, etc.)

In addition to code reuse across different register allocators which we obtain by basing a register allocator on the `Graph_Builder` and `Graph_Coloring` frame-

Module	Lines of Code
`Graph_Builder`	
Graph reconstruction	291
Graph construction	990
Total (Base Table 10)	1281
`Graph_Coloring`	
Coalescing	63
Color ordering	0
Color assignment	554
Spill-code insertion	394
Shuffle-code insertion	0
Misc	1532
Total (Base Table 9)	2543
Total	3824

Table 11: Size of register-allocation framework.

works, code is shared by the `Splitting_Style` and `Simplicafication_Based` classes. Both of these classes manipulate the interference graph through simplification. Simplification determines color ordering and is also used to find the splitting points (for register allocators derived from `Splitting_Style` [24]). The shared code for graph coloring/building of Table 12 is also organized as a framework. This structure allows different register-allocation approaches to use different heuristics to relax the graph constraints when the process blocks. For example, Chaitin-style coloring spills the least-cost (spill-cost/degree) live range, optimistic coloring chooses the least-cost live range but does not spill, fusion-style coloring splits live ranges rather than spilling. Each method inherits from "Base" and provides the implementation of what to do when the simplification process blocks. Table 12 lists the sizes of this shared code.

For example, the code for Chaitin-style coloring consists of `Graph_Coloring` (3824) (Table 11), `Simplification_Based` (40) and the "Misc" category (1859) of Table 9, `Conventional` (18) (Table 10), and the code for the `Simplification_Process` (155) with Chaitin-style heuristics (20) (Table 12). Given this frame-

Simplification Process	Allocator			Lines of Code
	[7]	[4]	[24]	
Base	x	x	x	155
Chaitin Style	x		x	20
Optimistic		x		16
Splitting			x	110
Spilling			x	60

Table 12: Shared code for graph building and coloring.

work, 67 % of the code is part of the common code base.

One of the benefits of this framework is that it allows us to evaluate different register-allocation heuristics easily. Moreover, a fair comparison among different approaches is now possible; for example, when some enhancement of a heuristic is added to `Simplification_Based`, all approaches based on simplification benefit from this improvement.

7 Management of multiple targets

Most compilers maintain a separate executable for each new target. This creates problems for version management and validation, as each target's compiler executable must be built and maintained separately. We avoid this problem by having a *fat compiler*, a single compiler executable that can generate code for any target. This can be done by dynamically selecting one of several implementations of a code generation interface, a strategy that has also been used explicitly by other projects[10], but that falls out of our frameworks-based implementation for free.

The class system for global data-flow analysis stabilized after the second revision. We are now at the third iteration of the code generator. Readers should not be discouraged by the number of iterations. Each iteration reduced the amount of code, a step we consider to be important for the long term. Also we fought at the same time with different C++ compilers. In the process of moving from one compiler to another, on a variety of platforms, we encountered more than one unpleasant surprise. (We used, with different success, g++ on Suns and Alphas, the DEC Alpha C++ compiler, SGI's C++ ("DELTA" compiler), MetroWerks Gold, and Symantec C++ (the last two on a 68K-based Macintosh).) Although it is hard to quantify the contribution of the changing compiler platforms to the development cost, we are beginning to see a stabilization of the environments.

8 Related work

Although there exists a large body of literature introducing application-specific frameworks, empirical studies are hard to come by. [14] describes experience for a different application domain (protocol software for the signalling module). Although compilers provide a nice domain for the use of object-oriented technology, there have been only a few reports. Most of these focus on front-end issues. [21] advocates a syntax-directed view of objects.

The use of application frameworks has been reported in the context of a Smalltalk compiler: [13] discusses front-end issues (parsing, semantic analysis), [16] presents a toolkit for constructing code optimizers for Smalltalk. It is interesting to contrast the objectives and decisions of [16] and `cmcc`. [16] aims to support the construction of a range of machine-specific optimizers; it is based on RTL (the Register Transfer Language). `cmcc` uses aggressive optimization technology in a machine-independent optimizer, register allocator, and instructions scheduler, and strives to keep the different code generators coupled, in the sense that they continue to benefit from improvements to the shared core compiler system. To the best of our knowledge, no empirical account of the impact of the framework technology on code reuse in the RTL system has been published. (Another difference is that [16] deals with Smalltalk and `cmcc` with C.) [15] includes a brief description of a simple object diagram compiler, and some of the patterns described in [12] are derived from translation systems. However, we are not aware of any investigation of frameworks for a compiler's global optimizer or retargetable back end, yet these components determine the success of an optimizing compiler.

9 Conclusions

The low-level objects to be used by an optimizing compiler are well understood — the usual data structures from an advanced undergraduate algorithms class go a long way, and there exist a number of class libraries for common data structures. However, the design of frameworks for global data-flow analysis, instruction scheduling, and register allocation as well as the set of abstract classes that form the foundation for the code generator, allow us to reuse code beyond low-level building blocks.

These frameworks contribute significantly to the organization of the compiler code. The code is clean and concise, easy to understand, and has been reused a large number of times: the code for the data-flow module has been reused 15 times, the code generator code is reused 3 times, and more than 75% of the code for a code generator is actually part of the common code base. The domain of optimizing compilers provides a further example for the power of the idea of application frameworks.

The abstract classes of the code generator play a crucial role in our plan to build a retargetable system, but their importance extends this aspect. These abstract classes allow us to make the compiler extensible. The code selector, register allocator, instruction scheduler, and code emitter are written to work on objects of type `Inst`, although the actual objects

are of type `MIPS_Inst`, `IWARP_Inst`, or `SPARC_Inst`. By using this one base class, we are now in a position to extend the compiler, for instance, to reorder phases or to add optimizations.

Another, although more indirect, metric to gauge the benefits of frameworks in this application domain is the mere fact that a group of 2.5 persons was able to construct a competitive compiler. (And constructing this compiler was not the only activity for this group.) Good advice on compiler construction is hard to come by — the `cmcc` project has demonstrated how modern programming technology can be employed by a small team to produce a high-quality optimizing compiler, in a short period of time.

References

[1] A. Adl-Tabatabai and T. Gross. Source-level debugging of scalar optimized code. In *Proc. ACM SIGPLAN'96 Conf. on Prog. Language Design and Implementation*, pages 33–43. ACM, May 1996.

[2] D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill code minimization techniques for optimizing compilers. In *Proc. ACM SIGPLAN '89 Conf. on Prog. Language Design and Implementation*, pages 258–263. ACM, July 1989.

[3] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[4] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. ACM SIGPLAN'89 Conf. on Prog. Language Design and Implementation*, pages 275–284. ACM, July 1989.

[5] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proc. ACM SIGPLAN'91 Conf. on Prog. Language Design and Implementation*, pages 192–203, Toronto, June 1991. ACM.

[6] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proc. ACM SIGPLAN 1982 Symp. on Compiler Construction*, pages 98–105, June 1982. In SIGPLAN Notices, v. 17, n. 6.

[7] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation by coloring. Research Report 8395, IBM Watson Research Center, 1981.

[8] F. Chow. *A Portable, Machine-independent Global Optimizer – Design and Measurements*. PhD thesis, Stanford University, 1984.

[9] F. C. Chow and J. L. Hennessy. A priority-based coloring approach to register allocation. *ACM Trans. on Prog. Lang. Syst.*, 12:501–535, Oct. 1990.

[10] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[11] S. Freudenberger and J. Ruttenberg. Phase ordering of register allocation and instruction scheduling. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, pages 146–170. Springer Verlag, 1992.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Object-Oriented Software*. Addison Wesley, 1995.

[13] J. Graver. The evolution of an object-oriented compiler framework. *Software: Practice & Experience*, 22(7):519–535, July 1992.

[14] H. Huni, R. Johnson, and R. Engel. A framework for networked protocol software. In *Proc. OOPSLA'95*, pages 358–369. ACM, 1995.

[15] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[16] R. Johnson, C. McConnell, and J. Lake. The RTL system: A framework for code optimization. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, pages 255–274. Springer Verlag, 1992.

[17] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *Proc. ACM SIGPLAN'92 Conf. on Prog. Language Design and Implementation*, pages 224–234. ACM, June 1992.

[18] J. Knoop, O. Ruthing, and B. Steffen. Lazy strength reduction. *J. of Prog. Languages*, 1(1):71–91, 1993.

[19] J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Trans. on Prog. Lang. Syst.*, 16(4):1117–1155, July 1994.

[20] J. Knoop, O. Ruthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN'94 Conf. on Prog. Language Design and Implementation*, pages 147–158. ACM, June 1994.

[21] K. Koskimies. Software engineering aspects in language implementation. In *Proc. 2nd Intl. Compiler-Compiler and High-Speed Compilation Workshop, Lecture Notes in Computer Science Vol. 371*, pages 37–51, Berlin, 1989. Springer Verlag.

[22] B. Lewis, L. Deutsch, and T. Goldstein. Clarity Mcode: A retargetable intermediate representation for compilation. In *Proc. ACM/SIGPLAN Workshop on Intermediate Representations (IR'95)*, pages 119–128, San Francisco, Jan 1995. ACM.

[23] J. Lewis, S. Henry, D. Kafura, and R. Schulman. On the relationship between the object-oriented paradigm and software reuse: An empirical investigation. *Journal of Object-Oriented Prog.*, 5(4):35–41, 1992.

[24] G. Lueh, T. Gross, and A. Adl-Tabatabai. Global register allocation based on graph fusion. Technical Report 96-106, Carnegie Mellon University, School of Computer Science, March 1996.

[25] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb 1979.

[26] S. Naeher et al. Leda, a library for efficient data types and algorithms. Avail. via FTP from sbsvax.cs.uni-sb.de, 1994.

[27] C. Norris and L. L. Pollock. Register allocation over the program dependence graph. In *Proc. ACM SIGPLAN '94 Conf. on Prog. Language Design and Implementation*, pages 266–277. ACM, June 1994.

[28] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[29] J. Ruttenberg, G. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heurisitic methods in a production compiler. In *Proc. ACM SIGPLAN'96 Conf. on Prog. Language Design and Implementation*, pages 1–11. ACM, May 1996.

[30] A. Stepanov and M. Lee. The standard template library. Tech. Report HP Lab. HPL-94-34, 1994. Available from http://www.cs.rpi.edu/-/ musser/stl.html.

[31] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

[32] Taligent Inc. Building object-oriented frameworks. http://www.taligent.com/building-oofw.html, 1994.

A Data-flow analysis details

Here we reproduce the details of the DFA framework from a technical report. A user familiar with this kind of framework may skip this section, it is included for the benefit of those readers without a background in compiler construction.

At the topmost level of the DFA class hierarchy is the `Data_Flow` class (Figure 1). Figure 2 highlights those member functions and fields that are of interest for this class.

The `GEN`, `KILL`, `IN` and `OUT` arrays hold pointers to bit vectors for each basic block and are indexed by basic block numbers. Bit vectors are allocated by the `init()` method. `fg_iter` is a generic iterator that iterates over basic blocks. This member is initialized by inheriting classes, to an iterator that iterates in the forward or backward direction in the control flow graph. The `confluence()` and `transfer()` methods perform the confluence operation and transfer functions at a basic block. `trans()` is a default transfer function that can be used by inheriting classes. The `init_Root()` method initializes the source or sink node's In or Out set, while the `init_In_Out()` method initializes all other nodes' In and Out sets. The `init_Gen_Kill()` method initializes a block's Gen and $Kill$ sets. `new_BV()` is a method that allocates a bit vector. This method is used by `init()` to allocate bit vectors to the bit vector arrays. `init()` must be called before the data-flow analysis is performed.

The fixed point computation is performed by the `dfa()` method; the definition of this method is shown in Figure 10. When the fixed point is computed, `propagate()` is called for each basic block. An inheriting class uses the results of the data-flow analysis in the `propagate()` method.

In the next level of the hierarchy, there are the `Fwd_Data_Flow` and `Bwd_Data_Flow` classes, which inherit from the `Data_Flow` class. Figure 11 shows `Fwd_Data_Flow`, the other class is symmetric. These two classes specialize the `Data_Flow` base class by defining the direction of the DFA, and thus define all the virtual functions in the `Data_Flow` class that depend on the direction of the DFA: `confluence()`, `init_Root()` and `transfer()`. (Note that the `transfer()` is kept as a `protected` member so that inheriting classes can override this method if necessary.) These classes also initialize the flow-graph iterator `fg_iter` by allocating an iterator that iterates in the appropriate direction. The `confluence()` method defined in the `Data_Flow` class depends both on the direction and confluence operation of the DFA. Since the confluence operation is not yet known, a new virtual function `confl_Op()` is defined. Note that since the direction of the DFA is known, we either initialize the In set of the source basic block or the Out set of the sink basic block. Therefore these classes define new virtual methods that initialize these sets: `init_Src_In()` and `init_Sink_Out()`.

The third level of the hierarchy adds the confluence operation to the direction, resulting in four classes ($\{\texttt{Fwd}, \texttt{Bwd}\} \times \{\texttt{Or}, \texttt{And}\}$); two of these are shown in Figure 12. These classes inherit from the `Fwd_Data_Flow` and `Bwd_Data_Flow` classes and define those virtual functions that depend on the confluence operation: `confl_Op()` and `init_In_Out()`. This leaves three virtual functions from the base `Data_Flow` class that must be defined by in-

```
void Data_Flow::dfa() {
    Block *block;
    //
    // initialize in, out, gen and kill sets
    //
    fg_iter->reset();
    while (block = fg_iter->next()) {
        init_Gen_Kill(block,GEN[block->id],KILL[block->id]);
        init_In_Out(IN[block->id],OUT[block->id]);
    }
    init_Root();
    //
    // compute the fixed point
    //
    int changed;
    do {changed = 0; fg_iter->reset();
        while (block = fg_iter->next())
            {confluence(block); changed += transfer(block->id);}
    } while (changed);
    //
    // Propagate results to basic blocks
    //
    fg_iter->reset();
    while (block = fg_iter->next()) propagate(block);
}
```

Figure 10: Fixed point computation.

```
class Fwd_Data_Flow : public Data_Flow {
    void  confluence(Block *b);
    void  init_Root() {init_Src_In(IN[src_block->id]);}
  protected:
    virtual void confl_Op(BV*,BV*) = 0;
    virtual int  transfer(int id)
        {return *OUT[id] = *trans(IN[id],GEN[id],KILL[id]);}
    virtual void init_Src_In(BV *bv) {bv->make_empty();}
  public:
    Fwd_Data_Flow(Block *src_blk)
        : Data_Flow((new Fwd_Block_Iter(src_blk))) {}
};

void Fwd_Data_Flow::confluence(Block *block) {
    Block *pred;
    foreach_pred_block(pred,block)
        confl_Op(IN[block->id],OUT[pred->id]);
}
```

Figure 11: The `Fwd_Data_Flow` class.

heriting classes: `init_Gen_Kill()`, `propagate()`, and `new_BV()`.

```
#define or_confl(in,out) *(in) += *(out)
#define and_confl(in,out) *(in) *= *(out)

class
Fwd_Or_Data_Flow : public Fwd_Data_Flow {
    void confl_Op(BV *in,BV *out) {
                              or_confl(in,out);}
    void init_In_Out(BV *in,BV *out)
        {in->make_empty();out->make_empty();}
  public:
    Fwd_Or_Data_Flow(Block *src_blk) :
             Fwd_Data_Flow(src_blk) {}
};

class
Fwd_And_Data_Flow : public Fwd_Data_Flow {
    void confl_Op(BV *in,BV *out) {
                            and_confl(in,out);}
    void init_In_Out(BV *in,BV *out)
        {in->make_full();out->make_full();}
  public:
    Fwd_And_Data_Flow(Block *src_blk) :
              Fwd_Data_Flow(src_blk) {}
};
```

Figure 12: The `Fwd_Or_Data_Flow` and `Fwd_And_Data_Flow` classes.

Figure 3 shows an example of how the DFA framework is used. This figure shows the class definition for the available expressions problem. In `cmcc`, this DFA is performed as part of partial redundancy elimination and its results are used to perform further analysis. Therefore the `propagate()` method performs no action. If the available expressions DFA was being used to eliminate common subexpressions, `propagate()` would be defined to iterate through the expressions inside a block and replace those expressions whose values are available with fetches from temporary variables. The `init_Gen_Kill()` method (whose code is not shown) simply iterates through the expressions inside a basic block, computing the block's gen and kill bit vectors. The `new_Expr_BV()` function (used in `new_BV()` to allocate a new bit vector) returns a bit vector suitable for holding a set of expressions.

A.1 Optimization internals

The DFA framework provides the foundation for the global optimization phase. The global optimization phase first transforms the loop structure of the program by partially peeling loops and inserting loop preheader blocks. After these control flow transformations, the optimizer performs constant propagation and folding using an algorithm based on abstract interpretation; this phase cleans up tests that were peeled out of loops. Assignment propagation and dead code elimination are based on the algorithms described in [8], and are repeated until they effect no more changes on the program. In the next step, the optimizer performs partial redundancy elimination using the algorithm described in [19]. Strength reduction is integrated with partial redundancy elimination using the algorithm described in [18]. Linear function test replacement and induction variable elimination are also performed during partial redundancy elimination. Finally, partial dead code elimination is performed using the algorithm described in [20]. Partial dead code elimination is repeated until no more changes are detected in the program.

B Code generator details

Figure 13 shows how conversion instructions are dealt with in the MIPS code generator.

```
struct MIPS_Convert_Inst : MIPS_Inst {
  public:
    enum Kind { cvtds,cvtdw, ... n_cvt };
                          // others ommitted
    const Kind kind;
  private:
    static const Info _Info[n_cvt];
  public:
    MIPS_Convert_Inst(Kind,Inst*);
    const Info& info() {return _Info[kind];}
};

MIPS_Convert_Inst::
           MIPS_Convert_Inst(Kind k,Inst *src)
    : MIPS_Inst(Inst::Convert,src), kind(k) {}

const MIPS_Inst::Info
MIPS_Convert_Inst::
          _Info[MIPS_Convert_Inst::n_cvt] = {
 {"cvt.s.d\t%x,%0","%r",MIPS_No_Imm,0,2,FP_REG},
 {"cvt.w.d\t%x,%0","%r",MIPS_No_Imm,0,2,FP_REG},
 ...
};
```

Figure 13: Providing the machine-specific information.

C Register allocator details

Figure 14 sketches the core engine of the register allocator.

```
class Graph_Coloring {
public:
    Graph_Builder *const builder;

    Graph_Coloring(char*, All_Physical_Regs*, Graph_Builder*, Code_Generator*);
    ~Graph_Coloring();
    void do_coloring(FGEdge_List&, BlockList&, Block_List&, Stack_Block&, Reg_Set&);
private:
    void color_assignment(Physical_Reg_Bank*, Vertex_Stack&, Vertex_Stack&);
    void spill_code_insertion(Block_List&, Stack_Block&);
    void coalescing(Block_List&);
    virtual void color_ordering(Physical_Reg_Bank*, Vertex_Stack&, Vertex_Stack&) = 0;
    virtual void shuffle_code_insertion(Block_List& blocks, Stack_Block& spill_block) {};
};

void Graph_Coloring::do_coloring(FGEdge_List& unionOrder,  Block_List& blocks,
                                 Stack_Block& spill_block, Reg_Set& reg_set) {
    // construct the interference graph
    builder->construction(unionOrder, blocks, this);

    int any_spill = 1;
    while (any_spill) {
        any_spill = 0;

        coalescing(blocks); // do live ranges coalescing

        Vertex_Stack COLOR, SPILL;
        Physical_Reg_Bank *bank;
        for(int i = 0; i < all_phyregs->n_banks; i++) {
            bank = all_phyregs->bank(i);
            // check if there are available registers for coloring
            if(bank->avail_regs() == 0) continue;
            COLOR.clear(); SPILL.clear();

            color_ordering(bank, COLOR, SPILL);
            color_assignment(bank, COLOR, SPILL);  // physical register assignment
            if (SPILL.length()) any_spill = 1;
        }
        if(any_spill)  // reconstruct the graph
            builder->reconstruction(blocks, all_phyregs);
    }
    spill_code_insertion(blocks, spill_block);
    shuffle_code_insertion(blocks, spill_block);
}
```

Figure 14: Details of `Graph_Coloring`.