Incorporating application semantics and control into compilation

Dawson R. Engler
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A
engler@lcs.mit.edu

Abstract

Programmers have traditionally been passive users of compilers, rather than active exploiters of their transformational abilities. This paper presents MAGIK, a system that allows programmers to easily and modularly incorporate application-specific extensions into the compilation process.

The MAGIK system gives programmers two significant capabilities. First, it provides mechanisms that implementors can use to incorporate application semantics into compilation, thereby enabling both optimizations and semantic checking impossible by other means. Second, since extensions are invoked during the translation from source to machine code, code transformations (such as software fault isolation [14]) can be performed with full access to the symbol and data flow information available to the compiler proper, allowing them both to exploit source semantics and to have their transformations (automatically) optimized as any other code.

1 Introduction

This paper presents MAGIK, a system that can be used to incorporate both application semantics and control into compilation. MAGIK is motivated by two sets of insights. First, programmer-defined data structures and functions define a semantically rich (albeit syntactically poor) language, built on top of the language the programmer uses to define them. Unfortunately, these *meta languages* have not had optimizers: optimization occurs at the lower-level of the programming language, but not at the high-level defined by their interface. Our belief is that since high-level operations are heavy-weight (e.g., they deal with file I/O, window manipulations, transactions, and thread creation), optimizations which understand their semantics offer the hope of significant speed improvements, potentially exceeding the impact of all other compiler optimizations. Second, programming has historically been passive: with the exception of restricted local code transformations provided by macro systems, programmers are limited to writing code, while the power to transform the code has been reserved for compilers. Our belief is that giving programmers safe, ready access to the compilation process will significantly improve the scope of programmer capabilities.

The MAGIK system has been built to test these beliefs. MAGIK provides a simple, modular mechanism for programmers to dynamically incorporate extensions into the MAGIK compiler. User extensions, written in ANSI C, are dynamically linked into MAGIK during compilation. Extensions are given access to MAGIK's intermediate representation (IR) through a set of interfaces that allow them to easily create, delete, and augment IR at compile time. Both this IR and MAGIK are built on top of the lcc compiler [3], which is used to compile the source language (ANSI C). The control MAGIK gives to programmers enables a broad class of optimization and code transformations. This paper presents ten such extensions and sketches of many more.

This paper concentrates on two abilities provided by MAGIK. First, it provides a way for implementors to include domain-specific semantics into compilation. Using this ability, implementors can build both interface optimizers (for speed) and interface checkers (for safety). Interface optimizers exploit application-specific knowledge in order to obtain performance improvements. Such optimizers are applicable to a wide range of interfaces: "bignums", message passing and I/O libraries, math libraries, matrix transformations for graphics, even simple queue operations. From a compiler perspective this ability is useful in any situation where providing a compiler "builtin" would allow more aggressive optimization. From an implementor perspective they are useful in situations where an interface implementor could look at a call or sequence of calls to his implementation and craft specialized call(s) that exploited local uses. For example, a file system implementor can write a optimizer that exploits knowledge of file system operations to perform optimizations such as hiding disk latency by both inserting disk block prefetching commands, and transforming synchronous file I/O operations into asynchronous ones. Interface checkers use application-specific knowledge to enforce stricter semantic checks. For example, by requiring that system call error codes be checked (or inserting such checks) or by ensuring that assertion conditions do not have side effects.

The second main ability MAGIK provides is an easy, modular way to do general code transformations with full access to source information. Using this ability, programmers can instrument code, augment it (e.g., by introducing software fault isolation code [14] or garbage collection reference counters) or enforce invariants about it (e.g., that no pointer casts are allowed). Unlike object code modifiers such as ATOM [11], MAGIK clients are tightly integrated with the source compiler. Performing transformations during the translation from high-level source language to machine code has two important characteristics. First, it provides access to the full semantics of the high-level language, information that source transformers can exploit (or require) during code transformation. Second, the IR produced from these user transformations is a first class citizen, optimized no differently than the IR produced by the compiler itself. As a result, the compiler optimizes these transformations as it would any other code.

This paper is organized as follows: Section 2 discusses related work. Section 3 provides an overview of the system. Section 4 provides two examples of incorporating user-level semantics into optimization, while Section 5 presents two transformations that exploit the information available at source level to augment the code. Section 6 presents further examples of how to use extensible compilation, Section 7 discusses issues in the current system and directions for future work and Section 8 concludes.

2 Related Work

Examples of including application-level information into compilation are compiler-directed prefetching and management of I/O [9] and ParaSoft's Insure++ [8], which can check for Unix system call errors (similar to the MAGIK checker shown in Figure 2). Using a MAGIK-based approach, systems such as these could be built without compiler modifications.

We compare MAGIK to macro systems, semantic-based optimizers, extensible compilers, and object code modifiers.

Macro systems are the most venerable instance of user-level code transformers. An advantage of such systems (Lisp is a good example) over MAGIK is their tight integration with the source language — extensions are typically written in the same language and style as the rest of the application. The main advantage MAGIK provides is power. Macro systems such as Weise and Crew's recent work [16] are restricted to fairly localized code transformations, while MAGIK extensions can perform global transformations across many interface calls, using symbol table and flow graph information provided by the compiler.

Mark Vandevoorde and John Guttag [12, 13] describe a system that provides programmers with a safe way to impart some classes of semantic information to the optimizer. User-level specifications for a restricted functional language are consumed by a theorem prover that optimizes based on the specific situation in which function calls are used. While their system is more automatic than MAGIK, it is less powerful. For instance, MAGIK gives programmers the ability to perform optimizations that appear difficult to express as specifications. The cost of this power is that MAGIK more difficult to use. Further practical experience is needed to determine if MAGIK's added power is worth this cost.

MAGIK follows in the footsteps of the Atom object code modification system [11] (foreshadowed by the object code modifiers of Wall [15] and Srivastava and Wall [10]), which provides users with the ability to modify object code in a clean, simple manner. Atom was one of the first tools to give programmers ready access to the transformational abilities encased in compilers. MAGIK complements this work, and trades the practical generality of dealing with object code for improved information and code efficiency gained by working within a high-level source compiler. Since MAGIK has access to all the information available to the source compiler (e.g., symbol table, flow graph information, high-level semantics) it can derive facts lost at the object code level. For instance, it can easily insert reference counts around all accesses to a particular pointer type; an object code modifier, working solely at the level of loads and stores, cannot. Furthermore, since MAGIK extensions are integrated with the optimization done by the compiler, they can be implemented more efficiently: IR added by an extension is optimized no differently than IR produced from source. In contrast, object code have to both work without much source-level information and cannot bootstrap existing compiler optimizers [15]. An important practical difference between MAGIK and object code modifiers is that MAGIK is significantly easier to implement. The system described in this paper took the author less than a month to implement and it runs on all targets that the base compiler supports (x86, Mips, Sparc). In contrast, duplicating the functionality of ATOM for even a single architecture would require significantly more work (especially on an architecture such as the x86).

There are many compilers designed to support easy addition of optimizations (e.g., SUIF [1]). These system could have been used to implement MAGIK; lcc was chosen because of the author's familiarity with it. To the best of our knowledge, none of these compilers have been used explicitly for extending the optimizer with user-level semantics or transformations.

MAGIK can be viewed as an "Open System" in the spirit of Kiczale's work [7].

Of course, programmers have long performed interface optimizations by hand. The advantages of automated optimization are well known.

3 System Overview

MAGIK provides a framework to extend compilation. User extensions are implemented as dynamically-linked functions. User extensions come in two classes: *code extensions* and *data structure extensions*. Code extensions are invoked at every function definition and are able to enumerate, add, delete, and modify MAGIK's IR as it makes the transition from source language to machine code. Data structure extensions are invoked at every data structure definition and are able to add, delete and modify structure elements. Since compiler internals are in flux, implementation portability is provided by isolating extensions from internal IR details via a set of standardized interfaces; multiple interfaces are provided, specialized to the main domains MAGIK is used in.

A given compilation may use many different extensions. To make the system usable, it is crucial that extension composition is modular. The two main requirements of modularity are that extensions be able to inspect the code produced by others and that extensions can be obliviously composed. MAGIK meets these requirements by providing three different extension types (*transformers*, *optimizers*, and *inspectors*) that correspond to the three main functional uses of extensions. Transformers are used to perform code transformations that do not depend on integration with global optimization (e.g., partially evaluating a C printf call). Optimizers are used to perform iterative optimization and are repeatedly invoked during global optimization until no IR modifications occur. (Optimizers differ from both transformers and inspectors in that they may be invoked multiple times.) Inspectors are similar in functionality to transformers except their placement in the extension pipeline ensures that they see all IR that will be compiled to code.

The main implementation limitation of the MAGIK system is that since lcc provides no global optimization framework optimizers are given only weak data flow information. We are investigating methods of removing this limitation (e.g., by using the SUIF compiler system [1]).

An operational overview of the extension process is as follows:

1. Programmers implement extensions using the MAGIK libraries; these extensions are compiled to object code.

The location of this code is either specified to MAGIK using command-line flags or by embedding the location in source files. For instance, header files can specify an extension to optimize the interfaces they define.

2. MAGIK compiles high-level source (ANSI C) to its internal IR in the traditional manner. As MAGIK encounters extension location directives (either as compiler flags or embedded in source) it uses the dld dynamic linker [5] to dynamically link the named extensions into the compiler proper.

3. At every function MAGIK encounters it invokes all code extensions, beginning with transformer extensions. At this point the extensions are free to augment, modify and delete parts of the IR. As part of the global optimization loop, MAGIK calls each optimization extension. These extensions have access to any data flow information computed by the compiler (e.g., use and def sets, values of procedure parameters, etc.) To ensure that code produced by any extension is visible to all others (a requirement for modular composition of different extensions) MAGIK loops through the extensions until no more modifications occur to the IR. A nice result of this organization is that the code produced by an extension is optimized as aggressively as the code produced from application source. After all optimization extensions have run, and no modifications occur, MAGIK runs inspector extensions in their specified order.

4. At every structure definition MAGIK invokes all structure extensions. These extensions can add, modify and delete structure entries. Typically these extensions are also paired with code extensions that augment data structure field uses and definitions.

5. MAGIK emits code.

MAGIK's lowest-level IR interface, based closely on that the the underlying compiler (lcc) is terse, simple and portable. Structurally, the IR is a tree language. Leaves are variables, labels, or constants; internal nodes represent operations performed on them (e.g., addition, indirection, jumps, function calls). When operands are created they are associated with a type selected from MAGIK's base types (shown in Table 1). Thereafter, types are implicit: operations infer their own types based on the type of their operands. Any conversions required by ANSI C are performed by MAGIK (e.g., as required by ANSI C a character variable will be converted to an integer before addition with an integer).

User-created IR (type: I_IR) is of a different type than native IR (type: X_IR). This distinction is helpful because user-constructed IR typically requires preprocessing before it can be sensibly incorporated into lcc's internal representation. By exploiting static type-checking, MAGIK can prevent users from blithely intermixing the different representations.

The interfaces are presented in the following tables: routines to allocate, lookup and manipulate symbols in Table 4, routines to construct IR in Table 5, and routines to navigate the IR in Table 2. Higher-level interfaces are discussed in Section 4 and Section 5.

We expect MAGIK to evolve with further experience. To aid iterative design, the current implementation has emphasized simplicity at all levels. MAGIK is built on top of the lcc retargetable ANSI C compiler [3], and uses its IR language as its fundamental interface [4] (higher-level interfaces are crafted on top of this). The regularity and small size of lcc's IR has been a major asset. Importantly, since mapping other IR's to the MAGIK IR and back should be straightforward, it can be realistically used as a basis for defining a standardized, compiler-independent, extension interface similar in availability to ANSI C's standardized libraries.

Type	C name
V	void
C	signed char
UC	unsigned char
S	signed short
US	unsigned short
I	int
U	unsigned
L	long
UL	unsigned long
F	float
D	double
P	void *

Table 1: MAGIK types (superset of lcc's types).

While the implementation exploits lcc's infrastructure, there is no fundamental tie to lcc. As experience with the system and its uses grows, reimplementations will occur in more aggressive compilers (or, alternatively, MAGIK will be used to enhance the optimization framework of lcc).

One of the common uses of MAGIK is to incorporate new functions as "built-ins" into the compiler. Since there can be tens or (at aggressive sites) hundreds of builtins, it is critical that the extension process itself is efficient. To achieve the required efficiency, MAGIK dynamically links extensions rather than isolating them in sub-processes that communicate via shared memory. In most cases this process has no significant impact on compilation speed. For implementations that wish to remove all overhead (at some cost in reduced flexibility) MAGIK provides an interface that can be used to statically link extensions into the compiler proper (similar to the process of adding device drivers to most operating systems).

The following two sections discuss the interfaces MAGIK provides for incorporating application semantics (e.g., interface optimization and checking) and for general code transformation.

4 Incorporating Application Semantics

As discussed previously, user-level data structures and functions define a high-level language, the semantics of which is unavailable to traditional compilers. MAGIK provides mechanisms that allow applications to construct extensions that can exploit these languages' semantics for improved semantic checking, optimization, and general transformations. The two main constructs of interest are functions and data structures. In the case of functions, clients are mainly interested in two pieces of data-flow information: the location of calls in relation to each other, and the definitions and uses of each call's operands and results. Clients also require semantic information about each call site's operands: their type, whether they are constants, and if so, what their values are. In the case of data structures clients are primarily interested in definitions and uses of structures and their fields.

To make IR manipulations easier, MAGIK exploits the limited information needed in this area to provide a default interface that is simpler than the general MAGIK IR. It includes basic block structures, function calls, details about function arguments and results (e.g., whether they are constants, their type, possible values, etc.) and information about structure accesses. A library of routines are provided that allow clients to add, modify, delete and augment function calls and code easily. Additional routines are provided to search for particular functions and lists of functions in the IR (easing IR navigation), traverse argument lists, and routines that compute the set of variables defined and used by a given call site. Table 6 presents MAGIK's interface for finding, manipulating, and constructing call sites. Table 7 presents MAGIK's interface for finding IR tree patterns, and both structure and structure field uses.

Clients that need access to the full power of MAGIK's IR can, of course, use it; the layering provided by default is intended as syntactic sugar rather than a barrier.

The following subsections present MAGIK's semantic interface and four example clients. The first client exploits MAGIK to perform the general transformation of adding a compiler "builtin" output function that is implicitly aware of its operand types (eliminating the need for printf-style format strings). The second client adds more rigorous semantic checking of Unix system calls by inserting checks around call sites that ignore a system call's return value. The third client ensures that signal handlers call only reentrant functions. Finally, the four extension optimizes RPC call sites by using partial evaluation to generate specialized argument marshaling code.

4.1 Example: adding type-aware functions

ANSI C suffers from the lack of a graceful mechanism to handle poly-typed functions. Programmers are typically reduced to specifying argument types using a manually-constructed type string. This methodology is clumsy and error prone. One of the more painful effects of this lack is that C is one of the few languages in use that does not have type-aware I/O routines.

Figure 1 presents a MAGIK extension that adds a type-aware output routine, output. It works by rewriting all calls to the poly-typed function it defines (output) to call printf using a type string (typestring) it constructs from the type of output's arguments. An operational view is as follows:

1. The extension iterates over all calls to output using the MAGIK functions FirstCall and NextCall.
2. For each callsite, it builds up a printf-style type string by iterating over output's argument list (using the MAGIK functions FirstArg and NextArg) and appending the type of each argument to typestring.
3. After typestring has been constructed, the extension uses RewriteCall to modify the call site to call printf instead of output and inserts typestring as the first argument.

A sample usage:

```
void example(int i, int j) {
  output("i = ", i, "j = ", j);
}
```

While some languages (such as C++) support this capability for simple scalers, our extension can be easily modified to print the fields in aggregate types, freeing programmers from having to tediously write data structure-specific output routines (this functionality was elided for brevity). Extensible code synthesis is powerful. Example uses include the automatic generation of routines to translate data structures between "in-core" and on-disk representations and the construction of linked-list, hash-tables, and associative arrays specialized to particular data structure types. A similar technique is used in Subsection 4.4 to construct an efficient argument marshaling routine for a remote procedure call system.

```
/* Add a type-aware output function. */
int RewriteOutput(X_IR c) {
#  define MAXARGS 64
   X_IR a;

  /* Foreach callsite, rewrite the output call. */
  for(c = FirstCall(c, "output");
      c != NULL;
      c = NextCall(c, "output")) {

    /* String to hold derived typestring. */
    char typestring[MAXARGS*2+1] = {0};

    /* Foreach argument, create a typestring. */
    for(a = FirstArg(c); a != NULL; a = NextArg(a)) {
      switch(OpType(a)) {
        case I: strcat(typestring, "%d "); break;
        case P:
          /* Print strings differently
               than pointers. */
          if(RawPtrType(NodeType(a)) == C)
            strcat(typestring, "%s ");
          else
            strcat(typestring, "0x%p ");
          break;
        /* ... */
        default: panic("Bogus type");
        }
    }
    /* Add newline */
    strcat(typestring, "\n");
    /* Change call to output to call to printf. */
    RewriteCall(c, "printf");
    /* Add typestring as printf's first argument. */
    PushArg(c, Cnststr(typestring));
  }
  return MAGIK_OK;
}
```

Figure 1: Routine to add a type-aware output routine to C

4.2 Example: safe system calls

C and Unix are notorious for using integer error codes to indicate exceptional conditions. C and Unix programmers are notorious for not checking these codes. This problem is a significant one, especially with the prevalence of network computing (where file I/O operations have to be retried with some frequency). Figure 2 presents an an extension that inserts

```
/* Add checks around unchecked Unix system calls. */
int RewriteUnix(X_IR c) {
  /* list of all calls we insert checks for */
  char *unixcalls[] = {"read","write","seek",/* ... */ 0};
  I_IR res, err, stmt;
  char *n;

  /* foreach callsite, rewrite the output call */
  for(res = NULL, c = FirstCallV(c, unixcalls);
      c != NULL; c = NextCallV(c, unixcalls)) {

    n = CallName(c);

    /* If anyone uses the result, assume it is checked. */
    if(Uses(c))
      continue;
    else
      warning("system call <%s> is not checked\n", n);

    /* Create a temp to hold syscall's return value */
    if(!res)
      res = Temp(inttype, MAGIK_REG);

    /* Create IR to assign the return value to res. */
    stmt = AddStmt(c, Asgn(res, ImportExprRef(c)));

    /* Create a call to error routine; it expects the
        syscall name, and the return code. */
    err = Call("error", voidtype, Cnststr(n), res, NULL);
    /* Insert check for syscall failure. */
    AddStmt(stmt, IfStmt(Lt(res, Cnsti(0)), err));
  }
  return MAGIK_OK;
}
```

Figure 2: Extension that places error checks around unchecked system calls.

error condition checks around unchecked Unix system calls and prints out errors that occur.

The extension works as follows:

1. It iterates over all calls to the functions listed in the array unixcalls using the MAGIK functions FirstCallV and NextCallV.

2. For each call site it checks if the result of the call is used (using the MAGIK routine Uses). Unfortunately, a use does not guarantee that the call's result is checked — for simplicity, we elide more aggressive checking.

3. For call sites that do not use the result of the system call, the extension creates IR to check the system call's return value and, if it is an error, call an error procedure (error) to print it out. It then inserts this IR into the original IR using AddStmt.

4.3 Example: safe signal handlers

Unix signal handlers represent primitive threads of controls. Unfortunately, they are used by many programmers who are unfamiliar with the dangers of threaded programs. A common mistake made is to call non-reentrant library functions from these handlers. If the application was suspended in the middle of a call to the same function (or to a function that manipulates state it depends on) the application program will, non-deterministically, exhibit incorrect behavior.

To help prevent this class of problems we have defined an extension that prevents calls to non-reentrant functions in a signal handler (the extension's code is elided for brevity). The extension works as follows:

- To trigger checking all signal handlers adhere to the naming convention of prefixing their name with "sig_" (e.g., sig_protection_fault).

- The extension scans for all functions beginning with this prefix and, for each callsite, checks that the call is either to one of a list of known reentrant functions or to a function that is prefixed with sig_. Any call that does not satisfy these requirements is flagged.

- To ensure that only checked handlers are installed as signal handlers it also looks for handler installation calls and checks that they only install functions beginning with the sig_ prefix.

4.4 Example: RPC specialization

Remote procedure call (RPC) is a widely used abstraction in distributed programming. A significant overhead of a general-purpose RPC call is the cost of copying the call's arguments into a message buffer ("argument marshaling"). Figure 3 presents a MAGIK extension that uses partial evaluation to remove the main contributor to this overhead, the interpretation of argument types, by crafting marshaling code specialized to a particular callsite.

The extensions infrastructure is similar to that used to implement output: it scans for calls to rpc and examines the its argument which, syntactically, is a call to a remote function. It decomposes this call into its constituent pieces and then builds marshaling code to copy each argument in the RPC call into a memory vector. It then rewrites the call to rpc to take a pointer to a local copy of the remote procedure along with a pointer to the constructed message buffer and its size. A sample usage is as follows:

```
int k,j,i;
double d;
/* ... */

/* call remote procedure remote_foo */
rpc(remote_foo(j, i, k, d));
```

Of course, this usage can be made prettier by communicating the names of remote procedures to the extension, thereby eliminating the need for the rpc annotation. For simplicity we do not perform this syntactic cleanup (we also ignore result passing).

5 Code transformations

Code transformations involve rewriting or augmenting general code (i.e., unlike the extensions described in the previous section, their domain is not limited to a specific interface). Example code transformations are software fault isolation, the translation of pointers from one representation to another, or the insertion of checks to ensure a pointer use is not nil.

In MAGIK, code transformations are typically implemented by searching for specific IR trees and (possibly) replacing or augmenting them. To make this style of usage easy, MAGIK provides an interface specialized to this domain. IR navigation can be implemented using MAGIK-provided pattern matching routines that iterate over IR, returning all locations that extension-specified IR trees occur at. Rewriting support includes procedures that insert, delete and augment IR subtrees. These routines isolate the programmer from implementation-specific details of IR modification (e.g., the need to update all pointers to a node that has been used as a CSE).

5.1 Example: structure packing

Dense structure layout can be used to improve locality. Figure 4 presents a data structure extension that rearranges structure fields to reduce structure size. Using the same capabilities extensions can perform many useful structure transformations: fields can be automatically arranged to be endian-neutral and on machines that lack sub-word operations, shorts and chars can be promoted to ints.

5.2 Example: software address translation

Software address translation is useful. It allows different addressing contexts to modularly and easily share of pointer-based data structures. It also supports easy "swizzling" where, for instance, the disk block pointers are translated to their incore equivalents.

A common trick programmers use to save space is to replace pointers with small integers. The optimization works by first defining a base offset (say the base of a memory arena that all data structures of a certain type are allocated out of) and then replacing pointers with small integers that record an object's relative offset from this base. Pointer reconstitution is performed by adding the offset and base together. While

```
/* Search for RPC calls and generate marshalling code. */
int MarshalGen(X_IR r) {
  /* foreach callsite, rewrite the output call */
  for(r = FirstCall(r, "rpc");
      r != NULL; r = NextCall(r, "rpc")) {
    I_IR index, marshalv;  /* marshalling vector. */
    int offset, sz;
    X_IR a, c;

    /* Allocate marshaling array on stack. */
    marshalv = Array(doubletype, Nargs(c));
    offset = 0;
    c = FirstArg(r); /* Remote call is first argument. */

    /* Store each argument in marshalling vector. */
    for(a = FirstArg(c); a != NULL; a = NextArg(a)) {
      /* ensure correct alignment. */
      offset = roundup(offset, NodeAlign(a));
      sz = NodeSize(a);

      /* Form expression "*(type *)(marshal + offset)"
           where type is typeof(a). */
      index = Index(
              Cast(Copy(marshalv), Ptr(NodeType(a))),
                 Cnsti(offset/sz));
      /* marshalv[offset] = a */
      PushStmt(c, Asgn(index, ImportExprCopy(a)));

      /* Add size of argument. */
      offset += NodeSize(a);
    }
    /* Replace rpc call with message send; send takes a
      pointer to a local copy of the remote function
      and the marshal vector and size as arguments. */
    c = ReplaceExpr(c,
        Call("send", inttype,
             CallName(c), Copy(marshalv),
                 Cnsti(offset), NULL));
  }
  return MAGIK_OK;
}
```

Figure 3: Extension that creates specialized marshaling code based on remote procedure call argument types.

```
/* Used by qsort to compare element sizes. */
static int pack_cmp(const void *p, const void *q) {
  return FieldSize(*(Field *)p) – FieldSize(*(Field *)q);
}

/* Look for structures with "pack_" prefix and
     minimize their storage size by sorting their
     elements by size.  */
void Packer(Symbol p) {
  unsigned n;
  Field *fl;

  if(strncmp(StructName(p), "pack_", 5) != 0)
     return;
  /* Get fields */
  fl = ImportFields(p, &n);
  /* Sort them. */
  qsort(fl, n, sizeof fl[0], pack_cmp);
  /* Write them out. */
  ExportFields(p, fl, n);
}
```

Figure 4: Routine to minimize structure size by sorting elements by alignment requirements.

profitable (especially on 64-bit machines) manual application of this optimization is, unsurprisingly, somewhat prone to errors. We present an extension that automatically performs this transformation. (*For time reasons we elide this example: it will be contained in the final paper.*)

6 Extensible compilation: patterns of use

This section delineates some broad classes of extensible compiler uses. Both simple and ambitious examples are included to give a flavor of the range of operations that can be performed. Many of the examples provide programmers with capabilities not previously available.

User semantic optimization As described in Section 4, the languages defined by interface's functions and data structures have not had optimizers that understood their semantics. Since the operations defined by these languages are heavy-weight, providing a mechanism to incorporate this information offers the potential of speed improvements exceeding the impact of all other compiler optimizations.

An example of this style of use is an extension that understands remote procedure call (RPC). When it encounters a series of RPCs, it can aggregate them into a single message (improving throughput) and by looking for definitions of their operands and uses of their results, replace synchronous RPC with asynchronous, and push the call higher in the program text, and the check for completeness right before any use (improving latency). Similar optimizations can be done for file I/O.

Another example is an extension that optimizes calls to a graphics library. Consider a sequence of calls that manipulate a matrix. Using a library-specific extension, it is possible to optimize across these calls, reusing intermediate results they compute, eliminating intermediate copies, and performing cache optimizations across them.

Finally, a "big num" package can optimize across calls to its operations.

Operationally, this approach can provide a performance gain for any situation where a system's implementor could look at a section of code and implement a specialized operation to capture the same functionality. The challenge with the extensions is to codify this knowledge.

Extension of compiler builtins Incorporating knowledge of functions into compilers in the form of "builtins" is profitable both in terms of syntactic sugar and in performance. Unfortunately, the inclusion of builtins requires the intervention (and interest) of compiler writers rather than system implementors. Consequently, it has been put to limited use despite its utility. Using MAGIK, implementors can easily add builtin procedures.

There are many simple routines (sorting, searching, tree and list manipulations) that are constantly reimplemented in order to work on different types. Using MAGIK these routines can be defined once, by an extension, and then used by all application writers. To illustrate this capability we have implemented a simple extension that defines a `max` procedure that works on any scaler argument type.

To show how MAGIK can be used to define builtin procedures for improved performance we have also written an extension that recognizes the ANSI C `memcpy` ("memory copy") function. The extension exploits information MAGIK provides to specialize to the local characteristics of each callsite. For example, in the general case, `memcpy` must treat its operands as unaligned. However, using the semantic information MAGIK provides, the extension can determine when a call site's pointer operands are aligned and specialize accordingly. Additionally, it unrolls and inlines the memory copying loop when the number of bytes to copy is a constant, Static specialization removes runtime selection overhead, and shrinks the function's `memcpy` footprint (due to the fact that the gaps introduced by non-taken cases is eliminated). These optimizations are profitable in the context of operating system device driver and networking code, which can extensively access fixed-sized quantities of partially unaligned memory.

Partial evaluation A more general form of builtin specialization is full partial evaluation. Using an extensible compiler, both automated systems and programmers construct partial evaluators for important routines. For example, Section 4 described an extension that generated specialized code for RPC marshaling.

Structure awareness The ability to automatically traverse, rearrange, redefine, and augment data structure members enables interesting operations. Data structure traversal allows the definition of structure independent routines for sorting, searching, marshaling, and printing. Control of data structure layout can improve performance by allowing extensions to group member fields that are used close together into the same cache line, improving cache behavior. It can also enhance usability by enabling extensions to abstract away such details as endianness by automatically rearranging structures to be endian neutral. Data structure redefinition can improve speed on machines that do not provide sub-word memory instructions by allowing an extension to replace sub-word sized structure

```
/* Look for function calls or assignments */
static int HasSideEffect(X_IR c) {
  if(!c)
    return 0;
  else if(Op(c) == ASGN || Op(c) == CALL)
    return 1;
  else
    return HasSideEffect(Left(c))
        || HasSideEffect(Right(c));
}

/* Check that assertions do not contain
    side–effecting optations. */
int AssertCk(X_IR c) {
  for(c = FirstCall(c, "assert");
      c != NULL;
      c = NextCall(c, "assert")) {

      /* The assertion expression is the call's
          first argument. */
      if(HasSideEffect(FirstArg(c)))
        warning("assert has a side–effect\n");
  }
  return MAGIK_OK;
}
```

Figure 5: Routine to guarantee that assertions are free of side-effects.

elements with word-sized ones. Data structure augmentation allows functionality enhancements such as automatic addition of bookkeeping fields needed by reference counting garbage collectors.

Added safety MAGIK offers improved software quality in addition to higher performance. Using it, implementors can construct checkers more stringent than provided by the compiler proper as well as inserting code to check for errors at runtime. For example, to ensure stronger pointer safety, MAGIK can be used to construct a code inspector that statically checks the IR generated at compile time to disallow all casts, implicit conversions, and adds runtime checks to guard against over and underflow of numbers, nil and bogus pointers, and out-of-bound array accesses. Figure 5 presents an extension that guards against side-effects in assertion macros.

The ability to insert integrity checks without requiring source modification is a powerful prophylactic measure to guard against errors, and can serve to elevate C (somewhat) to the realm of modern languages.

Passing compiler information to applications Compilers compute much useful information. MAGIK provides an infrastructure that can be used to pass this information to applications. Two example extensions we have built in this spirit are an extension that, given a pointer to a type, returns the alignment of that type (this is useful for memory allocators) and an extension that takes a single argument and indicates whether it is a constant expression (useful in making inline decisions).

Code transformations The ability to augment code is powerful. Using MAGIK's interfaces, applications can implement a vast set of code transformations such as the insertion of reference counting, software address translation (as described in Section 5), or providing protection via software fault isolation [14].

An interesting optimization is to encode the expected result of interface calls in an extension. These "annotations" allow the extension to rearrange code so that the conditional bodies of unexpected cases are moved off of the commonly executed path, thereby improving both instruction prefetching queue and instruction cache utilization.

Exploiting type information The ability to access symbol table information enables operations not typically supported in Algol languages. For example, programmers can use MAGIK to pass types as arguments to functions such as `malloc` so it can track the pointer type it is allocating and be accurate (rather than conservative) in the alignment it provides.

Investigation Ready access to a semantically-rich intermediate language can be used to answer many questions about source-level code. For example, it can be used to verify hypothesis about software engineering by correlating bug reports to how many times an abstraction layer is broken (perhaps by tracking structure accesses) or by correlating ease of modification to the number of intermodule dependencies a source file has. Checks can be inserted to check for the aliasing of pointers to determine what optimizations would be profitable. It can also be used to support graphical performance monitoring in the spirit of Jeffery and Griswold [6] by automatically inserting display calls around interface uses.

Other Uses There are many other uses for extensible compilation. For example, many uses of Atom can also be done using MAGIK(the tradeoff is less generality for more information and optimization). It provides an easy way to incorporate annotations into the optimization phase by looking for annotations in the form of function invocations. It can be used to restrict allowable operations in the input language in order to make it more amenable to optimization. Or it can be used augment the base language with abilities such as exception handling.

With a sufficiently rich intermediate language MAGIK's extension framework can be used to make its compiler into a truly open system, where a variety of implementors can augment its core optimization abilities with new optimizations. In this manner, compiler optimizations would become an order of magnitude easier to disseminate.

7 Discussion

MAGIK attempts to literally make "library design language design." It does this by attacking the three crucial differences between writing a function-level interface and defining an input language and compiler. The first difference is obvious: languages have syntactic sugar, libraries do not. By enabling interface designers to include context- and semantic-sensitive code transformers, sugar can be judiciously added to function interfaces (e.g., as done in the output and rpc examples in Section 4). The second difference is more subtle: languages allow semantic checks that can be difficult for a library to replicate in terms of its implementation language. By giving extensions access to both the symbol table and function-level IR this barrier can be eliminated. Finally, languages can be optimized. Encoding their semantics in a compiler allows a ready implementation of both local (e.g., peephole optimization) and global (e.g., CSE) optimizations. Current compilers are blind to interface semantics, precluding analogous optimizations. MAGIK provides mechanisms that can be used to build interface optimizers that optimize interface primitives as aggressively as source language constructs.

7.1 Interface issues

An interesting research question is determining the design rules for building interfaces that are amenable to language-like optimization techniques. Two principles seem relatively safe. First, high-level optimization is aided by the use of declarative, high-level interfaces that can then be "strength-reduced" to the characteristics of local usage. Second, optimization across interface calls is eased if the result of one interface call is immediately used by another: function call nesting is an ideal way of eliminating data-flow ambiguities. Thoroughly codifying practical precepts will be challenging.

Careful (but, unfortunately, iterative) design of the MAGIK system has allowed us build it so that it is integrated with the infrastructure lcc uses to construct its internal IR. An important result of this integration is that we have been able to use the frontend routines lcc provides for constructing abstract syntax trees. Using this code has two significant benefits. First, it allows users to only specify types when defining constants and symbols: the remaining IR-construction routines can derive required types from context (e.g., Add can determine it is an integer addition by examining its operands). Eliminating the need to explicitly encode types has dramatically simplified MAGIK's code construction interface. Second, lcc's routines are designed to perform implicit conversions as required by the rules for ANSI C. As a result, they type-check their arguments (providing users with safety) and perform coercions as necessary (providing users with convenience).

There are a few challenges to using the current IR system. The first is dealing with IR tree layouts across compiler versions. Layout of IR trees is a fairly volatile implementation feature. Currently, MAGIK decrees an IR interface and layout. The cost of this solution is that future implementations may require extra mapping code to compile their IR to the standardized MAGIK IR and back. An alternative solution is to specify code using a higher-level representation. The main technical challenge of using IR to specify patterns is that functionally identical language expressions may be compiled to structurally different trees. Fortunately, the lcc IR is spare enough that this problem is not difficult: the number of possibilities tends overwhelmingly to one and, in rare cases, two. In fact, the use of a low-level IR can have a significant benefit over both source-level and machine-code matching in this respect since both, in practice, can contain significant numbers of synonyms (e.g., consider the possible ways to get values to and from memory on the x86, or the different but equivalent methods to reference an array element in C). However, while the IR representation has been sufficient for all examples we've wanted to implement, there are times when a less strenuous mechanism of code specification is preferable. We are currently investigating alternatives.

7.2 System limitations

There are a number of limitations with the current system; most were deliberately chosen in order to allow it to be built quickly so that real programmers could use it in the near future, thereby allowing the wheel of iterative design to begin turning with the least amount of delay. Four main limitations are discussed.

First, constructing large pieces of code is tedious. This would naturally be remedied with language support. A promising avenue is to use the 'C language [2] (designed to construct code dynamically) as a sugary method of dynamically constructing MAGIK IR. 'C solves most of the semantic issues dealing with variable binding, and code construction, leaving us with the fairly straightforward task of modifying it to dynamically emit MAGIK IR rather than executable code.

Second, the current code specification MAGIK interface — the low-level IR of lcc — while simple, is perhaps not the most natural for mainstream programmers. There are tradeoffs in this representation: a low-level IR can be more precise, however, it can also be more complex than necessary. We are investigating the representation of code templates used for matching via language support: here to a modification of the 'C language seems promising.

Third, the system is manual, even for tasks that could be done automatically (e.g., in the spirit of Vandevoorde and Guttag [13, 12]). As we determine which of these tasks are important and common, automation will be added.

Finally, lcc, while simple and easy to modify, is a poor optimizer. We are examining ways to improve its code quality.

7.3 A simple language extension

Exploitation of application semantics is helped if semantics can be clearly and unambiguously indicated. For example, translating shared memory accesses is eased if every such access can be explicitly labeled as "shared." The clean, clear conveyance of semantic information to extensions is a general problem. Fortunately, it has a simple solution: the addition of a new syntax operation to ANSI C (annotation) that is used to create new, scoped type qualifiers. These qualifiers would be syntactically parsed and internally stored in the symbol table but otherwise ignored by the compiler proper — their semantics provided solely by extensions. An example usage:

```
/* add "shared" as a new type qualifier */
annotation shared;
/* Allocate an integer with new type qualifier. */
shared int x;
```

8 Conclusion

This paper has addressed two problems programmers have historically faced. First, the languages they define via interfaces have not been treated as first-class languages. As a result, these languages have had no language-specific semantic checkers, transformers, or optimizers. Second, their programs are passively consumed with little support for active transformation (such as rewriting of structure fields and the addition of profiling code).

The MAGIK system is a first step towards solving these problems. MAGIK provides a modular interface implementors can use to extend compilation. The main interaction is through a set of interfaces that give extensions access to the IR produced from source. MAGIK thus provides a method that system implementors use both to incorporate domain-specific semantics into compilation (thereby enjoying the obvious advantages of automated optimization and checking) and to perform general transformations on the IR produced from source (thereby having both access to high-level semantic information and the resulting transformation code optimized as aggressively as code produced from source).

This paper has presented many example clients of the MAGIK system. Many of these extensions provide capabilities that programmers did not previously have. Future research will involve both extending these capabilities and exploring their consequences.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[2] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg, FL, 1995.

[3] C. W. Fraser and D. R. Hanson. *A retargetable C compiler: design and implementation*. Benjamin/Cummings Publishing Co., Redwood City, CA, 1995.

[4] C.W. Fraser and D.R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(9):963–988, September 1991.

[5] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software—Practice and Experience*, 24(4):375–390, April 1991.

[6] Clinton L. Jeffery and Ralph E. Griswold. A framework for execution monitoring in Icon. *Software—Practice and Experience*, 24(11):1025–1049, November 1994.

[7] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[8] Adam Kolawa and Arthur Hicken. Insure++: A tool to support total quality software. `http://www.parasoft.com/insure/papers / tech.htm`.

[9] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.

[10] A. Srivastava and D.W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, March 1992.

[11] Amitabh Srivastava and Alan Eustace. Atom - a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[12] Mark T. Vandevoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, M.I.T., 1994.

[13] Mark T. Vandevoorde and John V. Guttag. Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity. In *Proceedings of the 1994 ACM/SIGSOFT Foundations of Software Engineering Conference*, 1994.

[14] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[15] D.W. Wall. Systems for late code modification. *CODE 91 Workshop on Code Generation*, 1991.

[16] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of PLDI '93*, pages 156–165, Albuquerque, NM, June 1993.

Operation	Description
X_IR LeftChild(X_IR n)	Returns n's left child or nil on error.
X_IR RightChild(X_IR n)	Returns n's right child or nil on error.
int OpType(X_IR n)	Returns opcode of n.
int Type(X_IR n)	Returns type of n.
int Align(X_IR n)	Returns alignment of n.
int Size(X_IR n)	Returns size of n.

Table 2: Base IR Interface.

Class	Examples	Prototype
Arithmetic binary operations.	ADD SUB MUL DIV XOR AND OR	I_IR op(I_IR a, I_IR b)
Arithmetic unary operations.	NEG COM	I_IR op(I_IR a)
Conversions ("convert to *type*").	CVTI CVTD CVTUS	I_IR *op*(I_IR a)
Memory operations.	ADDR INDIR	I_IR *op*(I_IR a)

Table 3: Partial IR-construction Interface. Functions determine the type of opcode to use based on operand type. Conversion conventions are those of ANSI C.

Operation	Description
I_IR Local(Type ty)	Creates a local variable of type t and returns its symbol.
I_IR LocalArray(Type ty, int n)	Creates a local array of type t and size n and returns its symbol.
I_IR Global(Type ty)	Creates a global variable of type t and returns its symbol.
I_IR GlobalArray(Type t, int n)	Creates a global array of type t and size n and returns its symbol.
I_IR Cast(I_IR var, Type t)	Creates a copy of symbol var changing its type to t.
I_IR Lookup(char *name)	Lookup symbol for variable name.

Table 4: Symbol construction and manipulation routines (routines to construct new aggregate types are elided).

Operation	Description
X_IR Copy(X_IR n)	Create a copy of node n. This function is typically used when adding a new subtree between a node and its child.
I_IR ImportExprRef(X_IR expr)	Import a reference to expr. This reference can then be used as an argument to functions that require a I_IR type.
I_IR ImportExprCopy(X_IR expr)	Import a copy of expr. This copy can then be used as an argument to functions that require a I_IR type.
X_IR AddStmt(X_IR a, I_IR stmt)	Add stmt after node a. Returns stmt.
X_IR PushStmt(X_IR a, I_IR stmt)	Add stmt before node a. Returns stmt.
X_IR DeleteStmt(X_IR stmt)	Remove stmt, returns its successor.
X_IR DeleteExpr(X_IR expr, I_IR replacement)	Delete node expr; replaces the tree with replacement. If replacement is nil, MAGIK will coalesce the tree expr was part of until it is well-formed.
X_IR AddExpr(X_IR a, I_IR b)	Insert b on top of a.
I_IR If(I_IR bool, I_IR stmt)	If bool is true, execute stmt.
I_IR IfElse(I_IR bool, I_IR stmt1, I_IR stmt2)	If bool is true, execute stmt1 otherwise execute stmt2.
I_IR While(I_IR bool, I_IR stmt)	While bool is true, execute stmt.

Table 5: Partial High-level IR construction Interface

Operation	Description
X_IR FirstCall(char *name)	Returns pointer to first call of name or nil if none is found.
X_IR FirstCallV(char **namelist)	Returns pointer to first call of any function in namelist or nil if none is found.
X_IR NextCall(X_IR c, char *name)	Returns pointer to next call of name or nil if none is found.
X_IR NextCallV(X_IR c, char **namelist)	Returns pointer to next call of any function in namelist or nil if none is found.
X_IR RewriteCall(X_IR call, char *newname)	Replace name of call to be newname.
X_IR FirstArg(X_IR call)	Return first argument (if any) of call.
X_IR NextArg(X_IR arg)	Get next argument (if any) after arg.
X_IR Arg(X_IR call, int n)	Returns the nth argument of call; returns nil on error.
void PushArg(X_IR call, I_IR arg)	Adds arg as the first argument to call.
void AppendArg(X_IR call, I_IR arg)	Adds arg as the last argument to call.
int NArgs(X_IR call)	Return number of arguments to call.
X_IR ReplaceArg(X_IR call, int argno, I_IR arg)	Replace argument argno in call with arg.

Table 6: Partial Function Navigation and Modification Interface

Operation	Description
X_IR Search(X_IR n, I_IR pattern)	Search for the tree pattern starting at location n. If n is nil, the search starts at the beginning of the function. Unspecified subtrees in pattern can be created using the function I_IR Any(Type ty).
X_IR FindStruct(X_IR n, char *StructName)	Search for use of StructName starting at n.
X_IR FindField(X_IR n, char *StructName, char *FieldName)	Search for use of field FieldName of type StructName starting at n.
Fields *ImportFields(Symbol p, unsigned *n)	Returns a pointer to an array of pointers to data structure p's fields. Elements in this vector can be reordered, deleted, and added.
Fields *ExportFields(Symbol p, Fields *fieldlist, unsigned *n)	Export fields (defined by fieldlist) as the layout for data structure p.
Field AddField(Symbol p, Field f1, Field f2)	Add field f2 after field f1 in structure p.
Field PushField(Symbol p, Field f1, Field f2)	Add field f2 before field f1 in structure p.
Field OverrideField(Symbol p, Field f, Type ty)	Change field structure p's field f type to ty.
Field FirstField(Symbol p)	Returns the first field in data structure p.
Field NextField(Symbol p, Field f)	Returns the next field in data structure p.

Table 7: Partial Structure Navigation and Modification Interface