Simple and Effective Link-Time Optimization of Modula-3 Programs

Mary F. Fernandez

November 7, 1994

1 Introduction

Object-oriented languages have features that help develop modular programs and libraries of reusable software. Opaque types and methods, two such features in Modula-3, incur a runtime cost, because to implement them, the Modula-3 compiler must generate code for various runtime computations and checks. Incomplete information at *compile* time necessitates these computations; their runtime overhead, however, can be reduced at *link* time when the entire program and its type hierarchy become available.

Opaque objects and methods are invaluable for developing libraries of reusable software. Opaque typing separates a type T's interface from its implementation and guarantees that clients that declare subtypes of T can be compiled even when the source that defines T's representation is unavailable, as is often the case for libraries. Opaque typing also supports smart recompilation: T's clients don't have to be recompiled when T's representation is changed. Modula-3 also supports overriding of methods, which permits a subtype of T to redefine any method inherited from T. The standard implementations of both features incur runtime costs. Accessing the field of an opaque type requires two loads: one to determine the field's runtime offset and a second to access the field. To support overriding, methods are implemented as indirect calls. The indirect call may be inexpensive — it might cost only one extra load to fetch the procedure's address — but it precludes other promising optimizations, such as procedure inlining and specialization.

One way to recover these runtime costs is to eliminate the features. The resulting design is similar to C++. The concrete representation of a C++ type T is revealed at compile time to T's clients. This revelation makes accessing a field of an object as efficient as accessing a field of a structure, but it increases recompilations because clients of T must be recompiled whenever T's representation changes. Also, C++ has both virtual methods, which may be overridden, and nonvirtual methods, which may not. Nonvirtual methods are implemented with direct calls, which saves a load and permits the compiler to inline methods in their clients. Inlining of nonvirtual methods also increases recompilations: if the source of a method changes, all clients that inline the method must be recompiled.

In this paper, we describe the opportunities for link-time optimization of Modula-3 and present two link-time optimization techniques. *Data-driven simplification* is a new technique. It uses a program's type

hierarchy to recover *completely* the cost of opaque types and to reduce the runtime overhead of methods. It also reveals other opportunities for optimization, such as constant and type propagation and procedure inlining and cloning. *Profile-driven* optimization uses profile data to identify and transform those procedures that can benefit most from optimizations made possible by data-driven simplification. Moreover, our techniques make it as easy to optimize procedures in libraries as to optimize those in applications.

Data-driven simplification and profile-driven optimization differ from other link-time optimizations because they require high-level data, e.g., the types of objects and expressions, that are necessary for applying our techniques but often missing from object code. Both optimizations require the entire program to be available and therefore cannot be applied at compile time. They are also machine independent and therefore are best applied to an intermediate representation before code generation. An intermediate representation also simplifies recognition of idiomatic expressions generated by the Modula-3 compiler. The compiler generates intermediate-code idioms to create objects, to access fields of opaque objects, and to invoke methods. Often, these idioms can be recognized and simplified. Identifying idiomatic expressions would be difficult for a traditional linker presented with object code that has been reordered by instruction scheduling or that does not carry enough type information.

Our optimization techniques are implemented in `mld`, a retargetable linker for the MIPS and SPARC. `mld` links `mill`, a machine-independent intermediate code that is suitable for link-time optimization and code generation. To evaluate the effectiveness of our techniques, we used `m3`, the DEC SRC Modula-3 v2.11 compiler [17], to compile five Modula-3 benchmarks. `m3` is not a native compiler: it generates C and invokes a C compiler to generate object code. To produce our results, `m3` invokes `mlcc`, an ANSI C compiler that generates `mill`. The benchmarks are non-trivial: they and the Modula-3 runtime system contain more than 170,000 lines of code. Although `mld` links Modula-3 programs, its implementation has few dependencies on Modula-3 itself and required only modest changes to the `m3` compiler (87 lines of new code) to produce the information needed by `mld`. `mld` could be used to evaluate link-time optimization of C++, for example, by using `mlcc` to compile C code generated by a C++ front end.

`mld`'s optimizations are simple and effective. Data-driven simplification is a local transformation and is inexpensive. The global optimizations applied during profile-driven optimization include constant and type propagation and procedure inlining and cloning, all simple and well-understood code transformations. Data-driven simplification reduces the total number of instructions executed by up to 11%, and it converts as many as 78% of the indirect calls executed to direct calls. Profile-driven optimization reduces the total number of instructions executed by up to 14% and the number of loads executed by up to 19%.

2 Opportunities for Link-time Optimization

An example best illustrates the opportunities for link-time optimization. A Modula-3 module is composed of an *interface*, which is included by clients of the module, and an *implementation*, which includes the module's

```
INTERFACE Hash;
TYPE HashT = OBJECT
 METHODS
  lookup(key: TEXT): REFANY;
  insert(key: TEXT;
         value: REFANY);
  delete(key: TEXT)
 END;
HashTab <: HashT
END Hash.
```

```
MODULE Symbol;
FROM Hash IMPORT HashTab;
TYPE SymbolTab = HashTab OBJECT
 level: INTEGER
 METHODS
  enterscope():= Enter;
  exitscope() := Exit
 OVERRIDES
  insert := SymInsert
 END;
 ...
END Symbol.
```

```
MODULE Hash;
REVEAL HashTab = HashT BRANDED OBJECT
 contents: ARRAY [1..101] OF REFANY
 OVERRIDES
  lookup := Lookup;
  insert := Insert;
  delete := Delete
 END;
 ...
END Hash.
```

Figure 1: Modula-3 Hash and Symbol Modules.

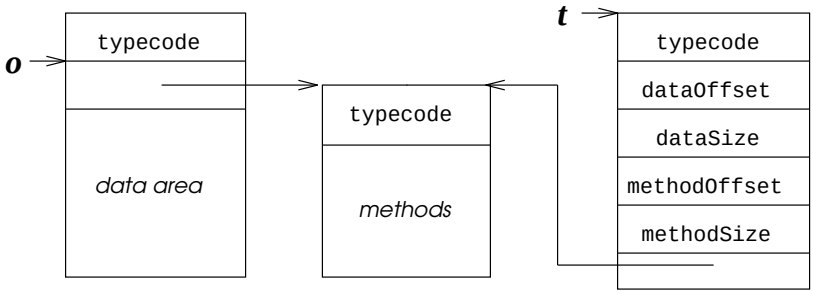
source code [19]. Fig. 1 gives the exported interface for the module `Hash` and the first few lines of the modules `Symbol` and `Hash`. The `Hash` interface (Fig. 1, left) exports `HashTab`, an opaque subtype of `HashT`; X `<:` Y declares X to be a subtype of Y. Because `HashTab` is a subtype of `HashT`, it has at least the same method suite as `HashT`: `lookup`, `insert`, and `delete`. Only these methods are accessible to its clients. The subtype relation does not reveal any more information about `HashTab` to clients or to the compiler; its private data and methods are hidden, and thus its underlying, or concrete, representation is unknown.

`Symbol` (Fig. 1, center) is a client of `Hash`. It declares `SymbolTab` as a subtype of `HashTab`, extending the definition of `HashTab` by defining the field `level` and the methods `enterscope` and `exitscope`. Even though `SymbolTab` is declared as a subtype of `HashTab` in `Symbol`, it is not necessary to recompile `Symbol` if `HashTab`'s representation changes, because `HashTab` is an opaque type. `SymbolTab` overrides `HashTab`'s definition of `insert`. Any invocation of `insert` by a `SymbolTab` object calls `SymInsert` instead of `HashTab`'s `insert`. `Hash` (Fig. 1, right) reveals the concrete representation of `HashTab` (but only within the `Hash` module). The revelation includes the declaration of `HashTab`'s private data (`contents`) and defines the values of its methods.

2.1 Implementation choices

Fig. 2 shows the runtime representation of Modula-3 types and objects. A type is represented at run time by a type descriptor, a C structure of type `TYPE`. In this paper, the runtime representation of `HashTab` is denoted by `HashTab_TC`. Objects are represented at run time by the type code of their type descriptor, their methods, and their data. Fig. 2 depicts an object o of type t. o contains a pointer to the methods field of its type descriptor (t`->methods`) and its own data area. t's typecode is duplicated in the type descriptor and its methods; t's methods are shared because they are immutable. After type initialization, the methods for *every* object of type t are the same, and all such objects share t's methods.

Link time is the earliest time at which the entire type hierarchy of a program is known. Without




```
typedef struct TC {
    int  typecode;
    int  dataOffset;
    int  dataSize;
    int  methodOffset;
    int  methodSize;
    ADDRESS  methods;
    ...
}  TYPE;
```

Figure 2: Object and Type Representations.

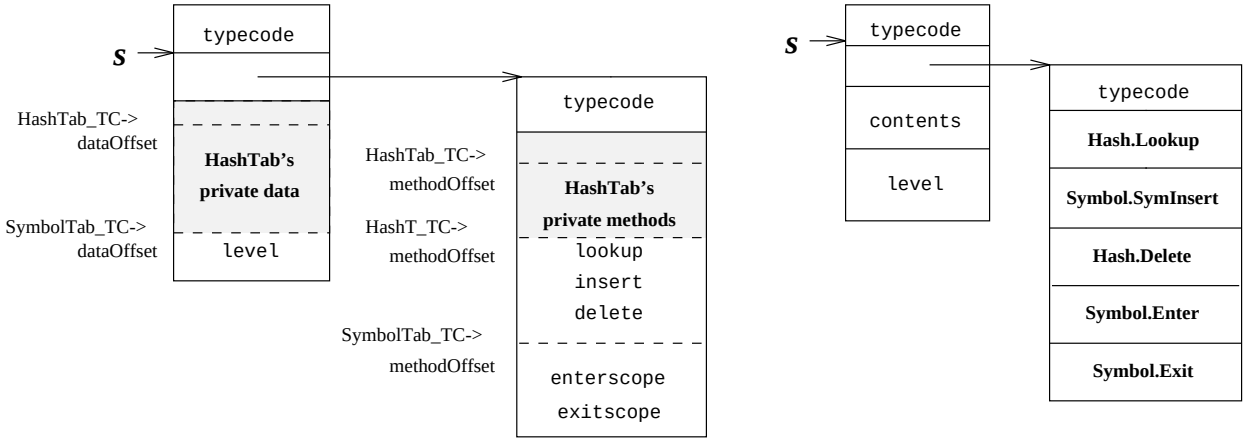


Figure 3: Compile-time and Run-time Representations of the `SymbolTab` Object `s`.

a linker that can use this information, program startup is the earliest point at which attributes of the concrete representations of all types, e.g., their sizes and the offsets to fields and methods, can be computed. Once initialized, these sizes and offsets remain constant. Fig. 3 depicts the compile-time and runtime representations (left and right pictures, respectively) of a `SymbolTab` object, `s`, in the module `Symbol`. The dotted lines in the compile-time picture indicate that the offsets to `s`'s fields and methods are unknown because `s`'s complete representation depends on the sizes of `HashTab`'s private fields and methods; the compiler knows nothing about the structure of the shaded areas. The values of `s`'s methods are also unknown at compile-time. These offsets and values are computed at program startup and are stored in the type descriptor. At run time, `SymbolTab` objects are represented as shown in the diagram on the right.

There are two sources of runtime overhead in the implementations of opaque typing and method invocation. To access fields and methods of opaque types, `m3` emits code to fetch the appropriate offset at run time. For example, in the expression `s.level`, the offset of `level` is unknown at compile time, so `m3` generates

```
((SymbolTab_fields *)(s + SymbolTab_TC->dataOffset))->level
```

where `SymbolTab_fields` is a C structure representing the private fields of `SymbolTab`. Similarly, the values

of `SymbolTab`'s methods are unknown, so the method invocation `s.lookup(key)` is compiled into

```
((HashT_methods *)((*s) + HashT_TC->methodOffset))->lookup(key);
```

where `(*s)` accesses the methods of `s`, and `HashT_methods` represents the public methods of `HashTab`.

`m3` implements method invocations as indirect calls to support overriding. In the last example, the invocation `s.lookup(key)` is compiled into an indirect call because the compiler does not know if `lookup` will be overridden in a subtype of `SymbolTab`. If `lookup` is overridden, then the runtime type of `s` determines which procedure is bound to `lookup`, i.e., `SymbolTab`'s `lookup` value or that of its subtype. Often this indirection is unnecessary. In this example, `lookup` is not overridden; it can only be bound to `Hash.Lookup`. The indirect call may be inexpensive: it might cost only two loads to fetch the procedure's address. It precludes other promising optimizations, however, such as procedure inlining and specialization, and performance may suffer on highly pipelined architectures where unpredictable branch targets stall the pipeline [22].

3 Data-Driven Simplification

`mld` uses data-driven simplification to simplify expressions that refer to variables whose values are known to be constant after linking. This technique is similar to partial evaluation and lets the linker simplify expressions that would otherwise be evaluated at run time. `mld` obtains the bindings between variables and their link-time values from a *binding* file, the contents of which resemble C assignments. A binding file contains assignments of values to globals. A value may be an integer, floating-point number, string, the contents of a structure or union, the value of another global variable, or an untyped block of initialized bytes.

For a Modula-3 program, the binding file is generated automatically by `mldpp`, a `mill`-code preprocessor, and contains the representation of the program's type hierarchy.[1] `mldpp` executes an initialization procedure similar to the one executed by the Modula-3 runtime system, but instead of initializing the type hierarchy in memory, it emits a binding file that describes the initialized type hierarchy.[2]

Fig. 4 gives part of the binding file produced by `mldpp` for the program in Fig. 1. The first assignment to `*HashT_TC` illustrates how `mld` uses binding statements. At link time, `mld` symbolically simplifies `mill` expressions that refer to variables defined in the binding file. For example, `HashT_TC->dataOffset` and `HashT_TC->methodSize` are replaced by `4` and `16`, respectively. Even though the runtime addresses of the type descriptors are unknown, the binding file specifies the *contents* of these structures. Initializing the type hierarchy at link time completely recovers the cost of referencing the fields of opaque objects. Every expression of the form $o + t\texttt{->}constfield + offset$ where o is an object address, t is the address of o's type descriptor, and *constfield* is a field of t bound to a constant at link time, is simplified to $o + C$ where C is $t\texttt{->}constfield + offset$. The simplified expression is as inexpensive as a structure field reference.

[1] A binding file is not restricted to type information; any "write-once" data is permissible, e.g., an array after initialization.

[2] A pre-processing step to compute the binding file is not necessary but helped in debugging; `mld` could compute this information on the fly before applying optimizations.

```
*HashT_TC = {
  typecode = 3;
  dataSize = 4;
  dataOffset = 4;
  methodSize= 16;
  methodOffset = 4;
  *defaultMethods = [
    # 4:  overridden by subtype
    # 8:  overridden by subtype
    # 12: overridden by subtype
  ];
  ...
};
```

```
*HashTab_TC = {
  typecode = 4;
  dataSize = 408;
  dataOffset = 4;
  methodSize= 16;
  methodOffset = 16;
  *defaultMethods = [
    4: Hash__Lookup;
    # 8: overridden by subtype
    12: Hash__Delete;
  ];
  ...
  };
```

```
*SymbolTab_TC = {
  typecode = 5;
  dataSize = 412;
  dataOffset = 408;
  methodSize= 24;
  methodOffset = 16;
  *defaultMethods = [
    4: Hash__Lookup;
    8: Symbol__SymInsert;
    12: Hash__Delete;
    16: Symbol__Enter;
    20: Symbol__Exit;
  ];
  ...
};
```

Figure 4: Example Binding File (# introduces comments).

The `mill` idioms for accessing fields of opaque types are generated by `mlcc` at compile time. `mld` uses `iburg` [10] to match and rewrite `mill` idioms at link time. The following two `iburg` rules match and rewrite the `mill` idioms for accessing fields of opaque types.

```
cfield:        INDIRI(ADDP(tc, const))            link_time_value(tc, const)
fieldaddr:     ADDP(ADDP(obj, const), cfield)     ADDP(obj, const + cfield)
```

Nonterminals are in lower case; `mill` operators are in uppercase. `tc` and `obj` match type descriptors and objects, respectively. `link_time_value` searches the binding file for the value at offset `const` in `tc`. `cfield` matches an expression whose constant value is bound at link time. `mill`'s intermediate code is based on the intermediate code used in `lcc` [9]. Of the 148 rules used by `mld` to simplify `mill` expressions, only 15 are specific to Modula-3. The others specify rules for constant folding, strength reduction, and for simplifying addressing expressions.[3]

3.1 Converting method invocations to direct calls.

The overhead of method invocation can be reduced by replacing method invocations with direct calls at link time. `mldpp` executes a conservative algorithm to identify methods that may be converted safely to direct calls. A method invocation $o.m$, where o is an object of type t, is convertible if t initializes method m's value to some procedure p or it inherits m's value from a supertype, and if m's value is not overridden in any subtype of t. If m is overridden, then it is impossible to convert conservatively an invocation of m because its procedure binding depends on the runtime type of the object that invokes m. If m is defined and not overridden, however, then exactly one procedure is bound to m in t and in all subtypes of t; any invocation

[3] Both `mlcc` and `mld` use the same set of simplification rules; most of them are never used by `mld` because `mlcc` simplifies most expressions at compile time. `mld` only uses the rules specific to Modula-3 and those for folding constants.

$o.m$ is guaranteed to invoke the *same* procedure.

The predicate $\mathit{convertible}(t, m)$ states the conditions for method conversion:

$$
\begin{aligned}
\mathit{convertible}(t,m) &= \mathit{has\text{-}method}(t,m) \wedge \neg \mathit{overridden}(t,m) \\
\mathit{has\text{-}method}(t,m) &= \mathit{initializes\text{-}method}(t,m,p) \vee \big(\exists u \text{ s.t. } \mathit{subtype}(t,u) \wedge \mathit{initializes\text{-}method}(u,m,p)\big) \\
\mathit{initializes\text{-}method}(t,m,p) &= m \text{ is initialized to procedure } p \text{ in the type declaration for } t \\
\mathit{overridden}(t,m) &= \exists s \text{ s.t. } \mathit{subtype}(s,t) \wedge \mathit{initializes\text{-}method}(s,m,p)
\end{aligned}
$$

In these definitions, t is a type; m is a method in the method suite of t; and $\mathit{subtype}(s, t)$ holds if s is a subtype of t. In addition to determining whether a method is convertible, `mldpp` computes its value; $\mathit{binding}(t, m)$, the procedure binding of m in t, is defined as

$$
\begin{aligned}
\mathit{binding}(t,m) &= \textbf{if } \mathit{initializes\text{-}method}(t,m,p) \textbf{ then } p \\
&\quad \textbf{else if } \mathit{has\text{-}method}(t,m) \textbf{ then } \mathit{binding}(\mathit{parent}(t),m) \\
&\quad \textbf{else } \text{Undefined}
\end{aligned}
$$

where $\mathit{parent}(t)$ is the immediate supertype of t. For each type t and method m such that $\mathit{convertible}(t, m)$ holds and $\mathit{binding}(t, m) = p$, `mldpp` emits a statement in the binding file assigning p to the offset of method m in t's methods. This assignment tells `mld` that an invocation of m may be converted to a direct call to p. For example, `SymbolTab` is a leaf type (it has no subtypes), so none of its methods are overridden. Each of its methods is also initialized, either via inheritance or in its type declaration, so each of `SymbolTab`'s methods is convertible. In Fig. 4, `SymbolTab`'s methods are initialized with the appropriate procedure value $\mathit{binding}(t, m)$; `mld` will convert invocations of these methods to direct calls to the corresponding procedures. `HashTab`'s `insert` method is not convertible because `SymbolTab` overrides the definition of `insert`; thus its value (at offset 8) is uninitialized.

Our technique for converting methods is largely language independent, but it does require that the linker recognize an idiom that depends on the language *and* on the specific implementation. For Modula-3, the idiom is $*o + t$`->methodOffset` $+$ *offset*. Other object-oriented languages and other implementations of Modula-3 might generate different idioms. The language dependencies in `mld` for recognizing method idioms are few: of `mld`'s 155 `iburg` rules for simplifying expressions, only 8 are specific to Modula-3.

3.2 Results

Table 1 summarizes the results of applying data-driven simplification to our benchmarks. Although there are only a few programs in our benchmark suite, none is trivial: they range from 3,000 to 81,000 lines of Modula-3.[4] The Modula-3 runtime system is more than 61,000 lines. Each program and the runtime system

[4] It is difficult to find non-proprietary applications in Modula-3 that use objects and libraries. We would like to find more such applications. Send mail if you have one.

Benchmark	Description (Lines)	Dynamic counts		
		% decrease instructions	% decrease loads	% decrease indirect calls
`interp`	A PostScript interpreter (19000)	4.0	6.8	13.5
`m3fe`	Analysis program in the `m3` toolkit (81543)	11.4	16.8	56.5
`prover`	A theorem prover (4536)	4.0	6.0	79.0
`pspec`	A program performance specification checker (9789)	2.6	3.8	21.4
`m3pp`	A Modula-3 pretty printer (3072)	3.0	4.0	1.7

Table 1: Results of applying data-driven simplification.

were compiled into `mill` with `m3` and `mlcc` and linked with `mld`. All measurements were taken on an unloaded DEC 5000, Model 240 running Ultrix V4.3 with 112 MB memory, a 64 KB direct-mapped instruction cache, a 64 KB data cache, and a local disk.

Data-driven simplification and method conversion are effective. Data-driven simplification reduces the number of instructions executed by 3–11%, the number of loads executed by 4–17%, and converts from 2–79% of the dynamic indirect calls to direct calls. We report exact instruction counts instead of elapsed execution times because instruction counts reflect precisely the effects of each transformation. Elapsed execution times are discussed in Section 4.

4 Profile-Driven Optimization

Data-driven simplification is a local transformation; it simplifies or eliminates expressions within a basic block. It reveals other opportunities for optimization, such as propagation of newly identified constants — e.g., the values of the fields `typecode`, `dataSize`, and `dataOffset` — and the type descriptors themselves. Revealing the complete type hierarchy also permits type expressions to be simplified. For example, `issubtype(`o, t`)` can be reduced to a constant boolean if the runtime type of object o can be propagated from its creation point to its uses. Because `mld` links the complete program, it could apply global transformations to every procedure, but dynamic program statistics indicate that this is unnecessary.

4.1 Dynamic program statistics.

Execution profiles of the benchmarks reveal that 2% or fewer of all procedures called execute at least 50% of all instructions executed. These procedures are the programs' "hot spots." In addition, more than 30% of all hot procedures are in libraries. Table 2 summarizes several dynamic statistics for the benchmarks. The second column gives the number of test inputs over which the data was acquired; the third gives the total number of procedures that were called during execution; and the fourth gives the smallest number of procedures that together account for at least 50% of the total execution time. The number of hot procedures

Benchmark	# Inputs	Procs called	Hot procs (lib)	% calls to hot procs
`interp`	42	1201	18 (6)	27.2
`m3fe`	197	4100	19 (8)	31.1
`prover`	3	915	13 (5)	31.7
`pspec`	2	1169	16 (5)	15.8
`m3pp`	56	624	11 (5)	26.5

Table 2: Dynamic statistics.

in libraries are parenthesized. Last, calls to the hot procedures account for a large percentage of total calls; the last column gives these percentages.

We must minimize the cost of global optimizations, because `mld` already pays for generating code for the entire program. These measurements indicate that broad application of global optimizations is unnecessary; applying global optimizations only to hot procedures and at frequently executed call sites will be both effective and inexpensive. To test this hypothesis, `mld` uses profiling data generated by QPT [2] to select hot procedures and then applies constant and type propagation to them. Our implementation of constant and type propagation uses standard iterative, data-flow analysis algorithms [1] applied to `mill` code. Many global optimizations could be effective when applied to the selected procedures; we chose constant and type propagation for two reasons. First, data-driven simplification reveals constant-valued expressions, which can be propagated, and the complete type hierarchy, which can be used to simplify type expressions. Second, our goal is not to introduce new global optimizations but rather to demonstrate the value of existing optimizations using information that is unavailable before link time.

4.2 Targeted inlining and cloning

Constant and type propagation often reveal useful information about procedures' calling contexts, e.g., constant-valued arguments or arguments whose runtime type is known. We apply *targeted inlining* and *targeted cloning* to procedures when their calling contexts are known. Targeted inlining attempts to inline the procedures at frequently executed call sites in hot procedures, then applies constant and type propagation in the caller and its inlined callees. `mld` chooses candidate sites using information gathered from profiles of the program's execution; a description of the technique is given below. Inlining is limited to those sites where it will not create too many locals, which is an approximate measure of register pressure.

Targeted cloning relies on the observation that procedures are often invoked with the same arguments from multiple sites. Instead of inlining and specializing a procedure at multiple sites, a copy of the procedure's text is made; the copy is specialized using the information about its arguments; and the clone is called in lieu of the original method. Targeted cloning applies constant and type propagation in hot procedures and their callers; if a call to a hot procedure or one of its callees has constant-valued arguments, the procedure is cloned

and is called in lieu of the original. The calling context of the original procedure is used to specialize the cloned version. Method cloning and specialization is a technique used in dynamically typed, object-oriented languages [5]; it improves run time at a small cost in space.

`mld` applies inlining at link time and chooses candidate sites using information gathered from profiles of the program's execution. Link-time inlining avoids many of the problems of source-to-source inlining: inter-module inlining does not create artificial dependencies between source modules, and library procedures can be inlined. `mld` initially chose call sites heuristically, but the inlined benchmark programs exhibited some of the typical problems programs with heuristically inlined callees: a few programs ran slower and link time increased significantly. `mld` now uses targeted inlining, a profile-driven technique that inlines at the program's most frequently executed call sites. Profile-driven inlining chooses call sites that are executed frequently and have the greatest chance of improving run time. Inlining is limited to those sites where it will not create too many locals, which is an approximate measure of register pressure. Other profile-guided inliners are used at compile-time and thus suffer the same problems as source-to-source inlining [6].

It is not effective to inline all calls to hot procedures or all calls in hot procedures because the same problems caused by heuristically guided inlining will occur. Instead, sites must be ordered according to a metric that indicates when inlining will be beneficial at the given site. Inlining ceases when the costs of negative secondary effects exceed the benefits of inlining. `mld` uses an *inlining factor* to determine where inlining might yield the greatest benefits. The inlining factor is a measure of the relative value of applying inlining to a particular call site and is computed from a summary of profiles generated by QPT.

`mld` reads a summary that consists of (*caller, callee, call site*) triples in decreasing order by inlining factor. For each call site, QPT reports the number of calls executed at the site (C) and the total number of instructions attributed to the callee when called from the given site (I). P is the number of instructions per call attributed to procedure call overhead. We assign P a machine-independent value of 1 instruction/call, which is conservative; the actual overhead depends on the architecture and the calling conventions. $P * C/I$ measures the ratio of call overhead to useful instructions executed by the callee. For example, if a site is called 5000 times and the callee executes 100,000 instructions when called from that site, a ratio of 0.05 is assigned to the site. The inlining factor $(P * C/I) * C$ weights each site by its overhead ratio and the number of times the call is executed. The metric favors those sites that have higher overhead-to-callee instruction ratios and are called more frequently than other sites. An arbitrary threshold limits the number of inlined sites: those sites with an inlining factor greater than 5000 are candidates. We have not yet evaluated the relative benefits and costs of choosing a particular threshold because they are machine-dependent. Not all candidates are inlined: if there are multiple candidates in a single caller, inlining is applied to the sites in decreasing order by inlining factor .

4.3 Results

Table 3 summarizes the results of applying both data-driven simplification and profile-driven optimization to our benchmarks. The static information provided in the table shows that application of global optimizations is limited: the maximum number of call sites inlined is 50; the maximum number of clones created is 200. The third column gives the number of call sites at which a clone is called and indicates that clones can be reused often, i.e., the calling context for a procedure is the same at multiple call sites. The italicized cases execute the fewest instructions.

Profile-driven optimization reduces the number of instructions executed by 4–14%, an additional 1–5% over data-driven simplification alone. At best, an additional 1.8% of the loads were eliminated. These improvements are more modest than we expected. One obstacle to producing better results is lossiness in `mill`. `mill` does not preserve as much type information as can be used by `mld`, because `mlcc`'s front end is based on that of `lcc`, which is designed for C, not Modula-3, and it discards some type information early in compilation. For example, the Modula-3 types for expressions that compute the addresses of elements in aggregate data structures (e.g., arrays and records) are not preserved. This prevents `mld` from propagating the types of objects in aggregate data structures and limits `mld`'s ability to simplify type expressions. A second obstacle is `mld`'s register allocator, which was not designed to handle procedures with inlined callees. `mld`'s code generators are based on those in `lcc`; `lcc`'s register allocator is machine independent and does not allocate across basic blocks, which can help reduce spills in frequently executed blocks. Both problems are limitations of our implementation and are not inherent to link-time optimization. Nonetheless, our results show that link-time optimization reduces the cost of methods and recovers the costs of opaque types. In addition, our techniques complement compile-time optimizations, because they leverage information unavailable before link time.

4.4 Elapsed execution times

Reductions in the number of instructions and loads executed are not reflected by comparable reductions in elapsed execution time. Elapsed execution times range from 8% slower to 20% faster than the baseline case, but there is no correlation between the number of instructions executed and run time. For the `prover` benchmark, one optimized version executes 9% *fewer* instructions than the baseline case but runs 3% *slower*; for the `pspec` benchmark, a version that executes only 3% fewer instructions runs 16% *faster*. We have determined that procedure placement is the cause of this anomaly. Both virtual-memory performance [25] and instruction and TLB miss ratios [21, 18] are known to be affected by procedure placement. For our benchmarks, poor (or better) instruction cache usage or an increase (or decrease) in TLB misses are possible explanations. We discount page faults and disk operations because their respective counts for the baseline and optimized versions of each benchmark are similar.

Appendix A gives elapsed execution times for various procedure layouts: execution times range from 9%

Name	Description
baseline	No transformations are applied.
data driven	Data-driven simplification applied using binding file.
	Data-driven simplification also applied in the following four cases.
inlining only	Applies inlining at active call sites, but does not apply global optimizations to the caller or its callees.
inlining+opts	Applies inlining at active call sites and global optimizations in the caller and inlined callees.
targeted cloning	Applies global optimizations to hot procedures, clones callees for which a calling context is known, but does not apply inlining.
all	Applies both "inlining+opts" and "targeted cloning".

	Static info			Dynamic info					
Benchmark	Inlined sites	Clones	Clone Sites	Total instructions	% dec.	Total loads	% dec.	Total indir. calls	% dec.
`interp`									
baseline				2,246,285,742		507,198,591		1,432,113	
data driven				2,155,521,196	**4.0**	472,765,679	**6.8**	1,238,835	**13.5**
inlining only	14			2,144,169,236	**4.5**	472,690,613	**6.8**	1,253,883	**12.4**
inlining+opts	14			2,136,445,167	**4.9**	471,296,993	**7.1**	1,249,592	**12.7**
targeted cloning		169	573	2,124,071,374	**5.4**	466,899,362	**7.9**	1,252,011	**12.6**
all	14	169	573	2,107,786,224	**6.2**	465,692,190	**8.2**	1,251,042	**12.6**
`m3fe`									
baseline				2,535,895,585		610,953,616		13,313,427	
data driven				2,247,994,707	**11.4**	508,081,278	**16.8**	5,787,760	**56.5**
inlining only	48			2,226,672,253	**12,2**	510,913,066	**16.4**	5,923,380	**55.5**
inlining+opts	48			2,224,026,335	**12.3**	509,093,838	**16.7**	5,928,671	**55.5**
targeted cloning		200	249	2,258,355,977	**10.9**	508,871,310	**16.7**	5,832,841	**56.2**
all	48	200	249	2,182,901,982	**13.9**	497,026,556	**18.6**	5,926,296	**55.5**
`prover`									
baseline				8,121,473,330		1,963,773,588		32,906,442	
data driven				7,796,928,277	**4.0**	1,845,354,851	**6.0**	6,906,463	**79.0**
inlining only	50			7,647,396,740	**5.8**	1,886,217,768	**3.9**	7,052,044	**78.6**
inlining+opts	50			7,580,044,906	**6.7**	1,893,356,427	**3.6**	6,960,740	**78.8**
targeted cloning		83	189	7,632,711,649	**6.0**	1,827,666,254	**6.9**	6,623,814	**79.9**
all	50	82	187	7,367,7293,86	**9.3**	1,848,698,243	**5.9**	6,692,476	**79.7**
`pspec`									
baseline				2,722,464,467		629,504,534		6,180,460	
data driven				2,652,447,345	**2.6**	605,755,797	**3.8**	4,859,839	**21.4**
inlining only	25			2,634,074,120	**3.2**	606,397,824	**3.7**	4,868,735	**21.2**
inlining+opts	25			2,626,013,632	**3.5**	604,426,754	**4.0**	4,818,071	**22.0**
targeted cloning		200	350	2,630,568,052	**3.4**	601,933,186	**4.4**	4,851,010	**21.5**
all	25	200	370	2,608,473,865	**4.2**	600,776,809	**4.6**	4,805,486	**22.2**
`m3pp`									
baseline				4,061,743,307		952,579,428		1,682,745	
data driven				3,939,194,472	**3.0**	914,111,621	**4.0**	1,658,992	**1.7**
inlining only	13			3,907,929,083	**3.8**	906,139,063	**4.4**	1,660,352	**1.3**
inlining+opts	13			3,895,297,753	**4.1**	906,697,230	**4.9**	1,657,298	**1.5**
targeted cloning		13	33	3,943,512,006	**2.3**	918,263,452	**3.6**	1,670,491	**0.7**
all	13	15	44	3,849,834,430	**5.2**	897,923,673	**5.7**	1,641,631	**2.4**

Table 3: Static and dynamic statistics

Modula-3			SPEC		
Benchmark	Link time	Text size	Benchmark	Link time	Text size
`m3pp`	29	281	`eqntott`	3.4	46
`prover`	50	454	`li`	6.5	99
`pspec`	46	563	`espresso`	13.6	232
`interp`	103	933	`gcc`	51.4	992
`m3fe`	185	1366			

Table 4: Link time in seconds and text size in KB

slower to 15% faster. Comparable variations in elapsed times due to code and data placement have been measured for SELF [15] when executing on a SPARCstation-2 with a unified, direct-mapped cache. Traditional linkers emit procedures in the order that they occur in modules; we rediscovered the effects of procedure placement because `mld` can emit procedures in various layouts, depending on the optimizations it applies. For our Modula-3 benchmarks, procedure positioning has even greater effects — both positive and negative — on run time than have been previously reported. Although this indicates that total instructions and loads executed are poor predictors of runtime performance for our benchmarks, the value of our techniques are simply masked, not invalidated, by the elapsed execution times.

5 System Performance

Linking intermediate code is slower than traditional linking because object code is generated for all modules every time they are linked. `mld` compensates by using BURS-based code generators [12, 20], which can select locally optimal code in as few as 50 instructions per tree node [11]. Nonetheless, code generation requires approximately 130μsec per generated instruction on a DEC 5000 and dominates `mld`'s total link time. Table 4 gives the time to link the Modula-3 benchmarks and four of the SPEC benchmarks. When linking the SPEC benchmarks on a DEC 5000, code generation accounts for 42–80% of the link time; for the Modula-3 benchmarks, it accounts for 48–58%. Seeking in large libraries and reading modules from them account for another 10%–35% of link time for the Modula-3 programs.

Fast compilation offsets slow linking. Because `mlcc` emits `mill` instead of assembly code, `mlcc` generates object code 1.5–3 times faster than the compiler on which it is based, `lcc`, and the MIPS assembler. Although `mlcc`'s speed cannot compensate for slower linking in an edit-compile-link cycle of only a few modules, it does compensate when every module is compiled and linked. For example, when compiling and linking all modules of the four integer SPEC benchmark programs, `mlcc` and `mld` are 1.2–1.5 times *faster* on a SPARCstation-2 and 1.1–1.9 times faster on the MIPS than `lcc` and `ld`; avoiding the assembler yields most of this improvement.[5]

[5] This comparison is unfair to the MIPS assembler, because the MIPS assembler schedules instructions but `mld` does not.

6 Related Work

We are not aware of any other linkers that link intermediate code to reduce the costs of high-level language features. There are other optimizing linkers, but they are intended for a family of related architectures [13, 27] or use machine-level representations such as register transfers [3, 23]. Their link-time optimizations tend to be architecture specific, e.g., rewriting object code to convert two-instruction address loads into one instruction loads on a 64-bit architecture [24].

Inlining is important for programs in which method invocations are calls. For dynamically compiled, object-oriented languages, such as SELF, inlining is essential for achieving acceptable runtime performance [5]. Even in statically typed languages, inlining is important enough to warrant explicit linguistic support. C++'s nonvirtual functions and the `inline` declaration [8] and Modula-3's `<*INLINE*>` directive are examples. When inlining is applied and how candidate sites are chosen most influence the benefits of inlining. For statically typed languages, inlining is often implemented as source-to-source transformations. Candidate call sites are chosen either explicitly by program directives or heuristically. Source-to-source inlining has disadvantages: inter-module inlining increases dependencies between source modules, and library procedures whose source is unavailable cannot be inlined. Choosing call sites heuristically also has disadvantages: aggressive heuristics may trigger inlining at numerous sites and increase compilation time significantly [7]; the number of local variables in a caller can increase significantly, which increases register pressure and spilling and hence execution time [7, 14]; and the heuristics are usually applied before the entire program is available, without knowing the execution frequencies of the inlined sites. Profile-guided inliners exist [6] but are used at compile-time and thus suffer the same problems as source-to-source inlining.

Converting method invocations to direct calls not only enables inlining but also may improve performance on highly pipelined architectures on which mispredicted or unpredictable branch targets stall the pipeline [22]. Accurately predicting the targets of indirect calls in C++ programs reduces pipeline stalls and is estimated to improve run times by 2–24% on an architecture similar to the DEC Alpha [4]. Eliminating indirect calls is even better, because identifying the target procedure requires no architecture-specific prediction mechanisms. A promising technique for converting method invocations is used in SELF [16]. At run time, the types of methods ("dynamically-dispatched calls") are fed back to the compiler to help it optimize call sites, e.g., by using type tests and direct calls for the most common cases and by inlining common targets. SELF is compiled on-the-fly, but the authors suggest that type feedback also could be effective for statically compiled languages. Link time might be a better target point for type feedback: type tests can be added and inlining applied without recompilation, and methods in libraries, for which source is often unavailable, can be optimized as well.

7 Discussion

High-level language features often incur costs that can be reduced or recovered when information about the type hierarchy becomes available. In C++, the concrete representation of a type is revealed to clients at compile time. This inclusion helps the compiler generate efficient code for accessing object fields and methods, but it reduces program modularity and complicates library maintenance by requiring recompilation of every client when the type's representation changes. In Modula-3, opaque typing enforces a strict separation between a type's interface and its implementation, which promotes program modularity and permits smart recompilation, but it incurs a runtime cost, because the compiler has insufficient information to generate efficient field-access code. Even though the representations of Modula-3 types are revealed at link time when the complete type hierarchy is known, traditional linkers are not designed to use this information and are unable to reduce the runtime overhead of these language features.

Our results show that link-time optimization of an intermediate code recovers the cost of opaque types and reduces the cost of methods while preserving the benefits of these features. Although we use `mld` to evaluate the effectiveness of link-time optimizations for Modula-3, little of its implementation is dependent on Modula-3, and the techniques themselves are language independent. `mld` could be used to evaluate link-time optimizations for C++; for example, data-driven simplification could provide C++ with the benefits of opaque types for no additional runtime cost, and link-time inlining of convertible methods could improve program modularity by reducing dependencies between source modules.

Intermediate-code linking is probably too expensive to use during edit-compile-test cycles because object code must be generated for the entire program, so fast turn-around is almost impossible. In its purest form, intermediate-code linking is too slow for use in development, but our results show that even some optimization at link time is valuable. Moreover, our techniques complement compile-time and other link-time optimizations because they leverage information unavailable to other optimizing tools. Possible techniques for reducing the cost of whole-program code generation include incremental linking of intermediate code or linking hybrid modules and libraries that contain some procedures in intermediate code and others in object code. Dynamic linking is even possible under certain conditions: for example, conservative method conversion is possible as long as no methods defined in dynamically linked modules override ones defined in statically linked modules. In future work, we plan to investigate techniques for making retargetable, optimizing linking a viable alternative to traditional linking during software development.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers – Principles, Techniques and Tools.* Addison Wesley, 1986.

[2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, NM, Jan. 1992.

[3] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7):329–338, July 1988.

[4] B. Calder and D. Grunwald. Call prediction in object-oriented languages. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 397–408, 1994.

[5] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–15, 1991. Published as *SIGPLAN Notices*, 26(11), 1991.

[6] P. Chang, S. Mahlke, and W. Chen. Profile-guided automatic inline expansion for C programs. *Software—Practice & Experience*, 22(5):349–369, 1992.

[7] K. D. Cooper, M. W. Hall, and L. Torczon. Unexpected side effects of inline substitution: A case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, 1992.

[8] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, Reading, MA, 1990.

[9] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, Redwood City, CA, 1995. ISBN 0-8053-1670-1.

[10] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sept. 1992.

[11] C. W. Fraser and R. R. Henry. Hard-coding bottom-up code generation tables to save time and space. *Software—Practice and Experience*, 21(1):1–12, Jan. 1991.

[12] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG—Fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, Apr. 1992.

[13] M. I. Himelstein, F. C. Chow, and K. Enderby. Cross-module optimizations: Its implementation and benefits. In *Proceedings of the Summer USENIX Technical Conference*, pages 347–356, Phoenix, AZ, June 1987.

[14] A. M. Holler. *A Study of the Effects of Subprogram Inlining.* PhD thesis, Department of Computer Science, University of Virginia, March 1991.

[15] U. Holzle. *Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming.* PhD thesis, Stanford University, August 1994.

[16] U. Holzle. Optimizing dynamically-dispatched calls with run-time type feedback. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–335, 1994.

[17] B. Kalsow and E. Muller. *SRC Modula-3 Version 2.11.* Digital Equipment Corporation Systems Research Center, January 1992.

[18] S. McFarling. Procedure merging with instruction caches. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 71–79, Toronto, Ontario, Canada, 1991.

[19] G. Nelson, editor. *Systems Programming with Modula–3.* Prentice Hall, 1991.

[20] E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 294–308, San Diego, CA, Jan. 1988.

[21] K. Pettis and R. Hansen. Profile guided code positioning. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1990. Also in SIGPLAN Notices, Vol. 25, No. 6, June, 1990.

[22] R. L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[23] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, pages 1–18, March 1993.

[24] A. Srivastava and D. W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 49–60, 1994.

[25] J. W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2), 1984.

[26] Standards Performance Evaluation Corp. *SPEC Benchmark Suite Release 1.0*, Oct. 1989.

[27] D. W. Wall. Experience with a software-defined machine architecture. *ACM Transactions on Programming Languages and Systems*, 14(3):299–338, 1992.

A Procedure placement

We expected that the reductions in the number of instructions and loads executed would be reflected by comparable reductions in elapsed execution time, but there is little correlation between the two. To determine if procedure placement is the cause of this anomaly, we emitted procedures for the baseline cases in three orders: as they occur in each module, by a pre-order traversal of the program's call graph, and by a post-order traversal. Profiling indicates that each version of each benchmark executes the same number and the same distribution of instructions, but elapsed execution times vary, ranging from 9% slower to 15% faster than the baseline. The table below gives the elapsed execution times of each benchmark in the three layouts. Each version of each benchmark had the same number of page faults and disk operations, and each run exceeded 90% CPU usage. For comparison, we compiled four of the SPEC [26] benchmarks with `mlcc` and used `mld` to produce the same three layouts of each benchmark. Here, procedure placement had little or no effect (0–2%) on the run times of the SPEC programs, which suggests that the effects of procedure placement are program specific.

Benchmark	By module	Post-order	Pre-order
`interp`	74.0	79.1	74.4
`m3fe`	105.1	113.2	114.4
`prover`	245.6	259.1	251.9
`pspec`	121.4	103.3	106.2
`m3pp`	112.8	115.2	109.0

Pettis and Hansen evaluated a number of profile-guided strategies for positioning code in executables. They found that a combination of positioning techniques result in runtime improvements of 2–26% and can be attributed to improved use of the instruction cache and reduced TLB misses. Procedure positioning was the least effective, improving run time by 0–10%; however, when procedure positioning was combined with basic-block positioning (moving the targets of frequently executed branches into fall-through positions) and procedure splitting (clustering frequently executed basic blocks on consecutive pages), significant improvements were measured. For our Modula-3 benchmarks, procedure positioning has even greater effects — both positive and negative — on run time. `mld` already uses profiling data to drive other optimizations, and it could be modified to do aggressive code positioning, but that would require some target-specific information to guide placement, e.g., cache and page characteristics.