Knit: Component Composition for Systems Software

Alastair Reid Matthew Flatt Leigh Stoller Jay Lepreau Eric Eide

University of Utah, School of Computing

{reid,mflatt,stoller,lepreau,eeide}@cs.utah.edu http://www.cs.utah.edu/flux/

Abstract

Knit is a new component definition and linking language for systems code. Knit helps make C code more understandable and reusable by third parties, helps eliminate much of the performance overhead of componentization, detects subtle errors in component composition that cannot be caught with normal component type systems, and provides a foundation for developing future analyses over C-based components, such as cross-component optimization. The language is especially designed for use with component kits, where standard linking tools provide inadequate support for component configuration. In particular, we developed Knit for use with the OSKit, a large collection of components for building low-level systems. However, Knit is not OSKit-specific, and we have implemented parts of the Click modular router in terms of Knit components to illustrate the expressiveness and flexibility of our language. This paper provides an overview of the Knit language and its applications.

1 Components for Systems Software

Software components can reduce development time by providing programmers with prepackaged chunks of reusable code. The key to making software components work is to define components that are general enough to be useful in many contexts, but simple enough that programmers can understand and use the components' interfaces.

Historically, developers have seen great success with components only in the limited form of libraries. The implementor of a library provides services to unknown clients, but builds on top of an existing, *known* layer of services. For example, the X11 library builds on the C library. A component implementor, in contrast, provides services to an unknown client while simultaneously importing services from an *unknown* supplier. Such components are more flexible than libraries, because they are more highly parameterized, but they are also more difficult to implement and link together. Despite the difficulty of implementing general components, an ever-growing pressure to reuse code drives the development of general component collections such as the OSKit [10].

Existing compilers and linkers for systems software provide poor support for components because these tools are designed for library-based software. For example, to reference external interfaces, a client source file must refer to a specific implementation's header files, instead of declaring only the services it needs. Compiled objects refer to imports within a global space of names, implicitly requiring all clients that need a definition for some name to receive the same implementation of that name. Also, to mitigate the performance penalty of abstraction, library header files often include specific function implementations to be inlined into client code. All of these factors tend to tie client code to specific library implementations, rather than allowing the client to remain abstract with respect to the services it requires.

A programmer can fight the system, and—by careful use of `#include` redirection, preprocessor magic, and name mangling in object files—manage to keep code abstracted from its suppliers. Standard programming tools offer the programmer little help, however, and the burden of ensuring that components are properly linked is again left to the programmer. This is unfortunate, considering that component interfaces are inherently more complex than library interfaces. Indeed, attempting to use such techniques while developing the OSKit has been a persistent source of problems, both for ourselves as developers and for OSKit users.

We have developed a new module language and toolset for managing systems components called *Knit*. Knit is based on *units* [8, 9], a model of components in the spirit of the Mesa [23] and Modula-3 [14] module languages. In addition to bringing state-of-the-art module technology to C programs, Knit provides features of particular use in the design and implementation of complex, low-level systems:

This research was largely supported by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, under agreement numbers F30602–99–1–0503 and F33615–00–C–1696. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

US Mail contact address: School of Computing, 50 S. Central Campus Drive, Room 3190, University of Utah, SLC, UT 84112–9205.

- Knit provides automatic scheduling of component initialization and finalization, even in the presence of mutual dependencies among components. This scheduling is possible because each component describes, in addition to its import requirements, specific initialization requirements.
- Knit's constraint-checking system allows programmers to define domain-specific architectural invariants and to check that systems built with Knit satisfy those invariants. For example, we have used Knit to check that code executing without a process context will never call code that requires a process context.
- Knit can inline functions across component boundaries, thus reducing one of the basic performance overheads of componentization and encouraging smaller and more reusable components.

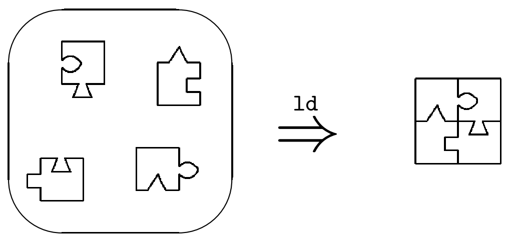
We have specifically designed Knit so that linking specifications are static, and so that Knit tools can operate on components in source form as well as compiled form. Although dynamic linking and separate compilation fit naturally within our core component model, our immediate interests lie elsewhere. We are concerned with low-level systems software that is inherently static and amenable to global analysis after it is configured, but where flexibility and assurance are crucial during the configuration stage.

Knit's primary target application is the OSKit, a collection of components for building custom operating systems and extending existing systems. Knit is not OSKit-specific, however. As an additional example for Knit, we implemented part of MIT's Click modular router [25] in terms of Knit components, showing how Knit can help express both Click's component implementations and its linking language.

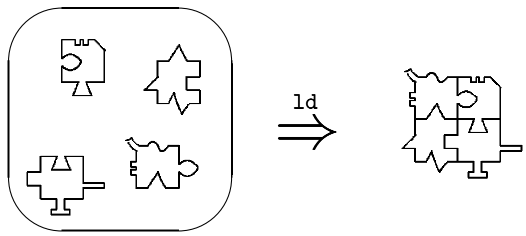
In the following sections we explain the problems with existing linking tools (Section 2) and present our improved language (Section 3), including its constraint system for detecting component mismatches (Section 4). We describe our initial experience with Knit in the OSKit and a subset of Click (Section 5). We then describe our preliminary work on reducing the performance overhead of componentization (Section 6). Finally, we describe related work (Section 7).

2 Linking Components

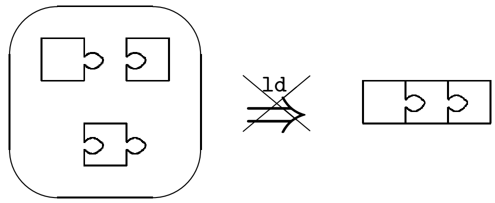
With existing technology, the two main options for structuring component-based, low-level systems are to implement components as object files linked by `ld` (the standard Unix linker), or to implement components as objects in an object-oriented language (or, equivalently, COM objects). Neither is satisfactory from the point of view of component kits; the reasons given in Section 2.1 and Section 2.2 reflect our experience in trying each within the OSKit. Though our experience with standard linking may be unsurprising, our analysis helps to illuminate the parts of our Knit linking model, developed specifically for component programming, described in Section 2.3.



(a) Typical system, expert's imagination



(b) Typical system, actual shape



(c) Interposition: `ld` rejects ambiguous tabs

Figure 1: Linking with `ld`

2.1 Conventional Linking

Figure 1(a) illustrates the way that a typical program is linked, through the eyes of an expert who understands the entire program. Each puzzle piece in the figure represents an object (`.o`) file. A tab on a puzzle piece is a global variable or function provided by the object. A notch in a puzzle piece is a global variable or function used by the object that must be defined elsewhere in the program. Differently shaped tabs and notches indicate differently named variables and functions.

The balloon on the left-hand side represents the collection of object files that are linked to create the program. The programmer provides this collection of files to the linker as a grab bag of objects. The right-hand side of the figure shows the linker's output. The linker matches all of the tabs with notches, fitting the whole puzzle together in the obvious way.

This bag-of-objects approach to linking is flexible. A programmer can reuse any puzzle piece in any other program, as long as the piece's tabs and notches match other pieces' notches and tabs. Similarly, any piece of the original puzzle can be replaced by a different piece as long as it has the same tabs and notches. Linkers support various protocols for "overriding" a puzzle piece in certain bags (i.e., archive files) without having to modify the bag itself. In all cases, the tedious task of matching tabs and notches is automated by the linker.

The bag-of-objects approach to linking has a number of practical drawbacks, however. Figure 1(b) illustrates the same linking process as Figure 1(a), but this time more realistically, through the eyes of a programmer who is new to the program. Whereas the expert imagines the program to be composed of well-defined pieces that fit together in an obvious way, the actual code contains many irrelevant tabs and notches (e.g., a "global" variable that is actually local to that component, or a spurious and unused `extern` declaration) that obscure a piece's role in the overall program. Indeed, some edges are neither clearly tabs nor clearly notches (e.g., an uninitialized global variable might implement an export or an import). If the new programmer wishes to replace a piece of the puzzle, it may not be clear which tabs and notches of the old piece must be imitated by the new piece, and which tabs and notches of the old piece were mere implementation artifacts.

The bag-of-objects approach also has a significant technical limitation when creating new programs from existing pieces. Figure 1(c) illustrates the limitation: a new component is to be interposed between two existing components, perhaps to log all calls between the top-right component and the top-left component. To achieve this interposition, the tabs and notches of the bottom piece have the same shapes as the tabs and notches of the top pieces. Now, however, the bag of objects does not provide enough linking information to allow `ld` to resolve the ambiguous tabs and notches. (Should the linker build a two-piece or a three-piece puzzle?) The programmer will be forced to modify at least two of the components (perhaps with preprocessor tricks) to change the shape of some tabs and notches.

2.2 Object-Based Linking

At the opposite end of the spectrum from `ld`, components can be implemented as objects in an object-oriented language or framework. In this approach, the links among components are defined by arbitrary code that passes object references around. This is the view of components implemented by object-based frameworks such as COM [22] and CORBA [26], and by some languages such as Limbo [6], which relies heavily on dynamic (module-oriented) linking.

Although linking via arbitrary run-time code is especially flexible, it is too dynamic for most uses of components in systems software. Fundamentally, object-oriented constructs are ill-suited for organizing code at the module level [7, 30]. Although classes and objects elegantly express run-time concepts, such as files and network connections, they do not provide the structure needed by programmers (and analysis tools) to organize and understand the static architecture of a program.

Symptoms of misusing objects as components include the late discovery of errors, difficulty in tracing the source of link errors, a performance overhead due to virtual function calls, and a high programmer overhead in terms of manipulating reference counts. Code for linking components is intermingled with regular program statements, making the code difficult for both humans and machines to analyze. Even typechecking is of limited use, since object-based code uses many dynamic typechecks (i.e., downcasts) to verify that components have the expected types, and must be prepared to recover if this is not so. These problems all stem from using a dynamic mechanism (objects) to build systems in which the connections between components change rarely, if ever, after the system is configured and initialized.

In short, object-based component languages offer little help to the programmer in ensuring that components are linked together properly. While objects can serve a useful and important role in implementing data structures, they do as much harm as good at the component level.

2.3 Unit Linking

The linking model for *units* [8,9] eschews the bag of objects in favor of explicit, programmer-directed linking. It also avoids the excessive dynamism and intractable analysis of object-based linking by keeping the linking specification separate from (and simpler than) the core programming language. The model builds on pioneering research for component-friendly modules in Mesa [23], functors in ML [21], and generic packages in Modula-3 [14] and Ada95 [18].

Linking with units includes specific linking instructions that connect each notch to its matching tab. The linking specification may be hierarchical, in that a subset of the objects can be linked to form a larger object (or puzzle piece), which is then available for further linking.

Unit linking can thus express the use pattern in Figure 1(c) that is impossible with `ld`. Furthermore, unlike object-based linking, a program's explicit linking specification helps programmers understand the interface of each component and the role of each component in the overall program. The program's linking hierarchy serves as a roadmap to guide a new programmer through the program structure.

Unit linking also extends more naturally to cross-component optimization than do `ld` or object-based linking. The interfaces and linking graph for a program can be specified in advance, before any of the individual components are compiled. The compiler can then combine the linking graph with the source code for surrounding components to specialize the compilation of an individual component. The linking hierarchy may also provide a natural partitioning of components into groups to be compiled with cross-component optimization, thus limiting the need to know the *entire* program to perform optimizations.

The static nature of unit linking specifications makes them amenable to various forms of analysis, such as ensuring that components are linked in a way that satisfies certain type and specification constraints. For example, details on the adaptation of expressive type languages (such as that of ML) to units can be found in Flatt and Felleisen's original units paper [9]. This support for static analysis provides a foundation for applying current and future research to systems components.

3 Units for C

In this section we describe our unit model in more detail, especially as it applies to C code. We first look at a simplified model that covers component imports, exports, and linking. We then refine the model to address the complications of real code, including initialization constraints.

3.1 Simplified Model

Our linking model consists of two kinds of units: *atomic units*, which are like the smallest puzzle pieces, and *compound units*, which are like puzzle pieces that contain other puzzle pieces. Figure 2 expands the model of a unit given in Section 2.3 to a more concrete representation for a unit implemented in C.[1] According to this representation, every atomic unit has three parts:

1. A set of *imports* (the top part of the box), which are the names of functions and variables that will be supplied to the unit by another unit.

2. A set of *exports* (the bottom part of the box), which are the names of functions and variables that are defined by the unit and provided for use by other units.

3. A set of top-level C declarations (the middle part of the box), which must include a definition for each exported name, and may include uses of each imported name. Defined names that are not exported will be hidden from all other units.

The example unit in Figure 2 shows a component within a Web server, as it might be implemented with the OSKit. The component exports a `serve_web` function that inspects a given URL and dispatches to either `serve_file` or `serve_cgi`, depending on whether the URL refers to a file or CGI script.

Atomic units are linked together to form compound units, as illustrated in Figure 3. A compound unit has a set of imports (the top part of the outer box) that can be propagated to the imports of units linked to form the compound unit. The compound unit explicitly specifies how imports are propagated to other units; these propagations can be visualized as arrows. A compound unit also has a set of exports (the bottom part of the outer box) that are drawn from the exports of the units linked to form the compound unit. The compound unit explicitly specifies which exports are to be propagated. Because all connections are *explicitly* specified, arrows can connect imports and exports with different names, allowing each unit to use locally meaningful names without the danger of clashes in a global namespace.

The imports of the linked units that are not satisfied by imports of the compound unit must be satisfied by linking them to the exports of other units within the compound unit. As before, the compound unit defines these links. The units linked together in a compound unit need not be atomic units; they can be compound units as well.

The example in Figure 3 links the previous example unit with another unit that logs requested URLs. The original `serve_web` function is wrapped with a new one, `serve_logged`, to perform the logging. The resulting compound unit still requires `serve_file` and `serve_cgi` to be provided by other units, and also requires functions for manipulating files. The compound unit's export is the logged version of `serve_web`.

3.2 Realistic Model

To make units practical for real systems code, we must enhance the simple unit model in a number of ways. Figure 4 shows a more realistic model of units in Knit.

First, instead of importing and exporting individual function names, Knit units import and export names in *bundles*. For example, the `stdio` bundle groups `fopen`, `fprintf`, and many other functions. Grouping names into bundles makes unit definitions more concise and lets programmers define components in terms of standardized bundles.

Second, the simplified model shows source code inlined in the unit's definition, but it is more practical to

[1] Knit actually relies on a textual language for unit descriptions, as shown in Section 3.3.

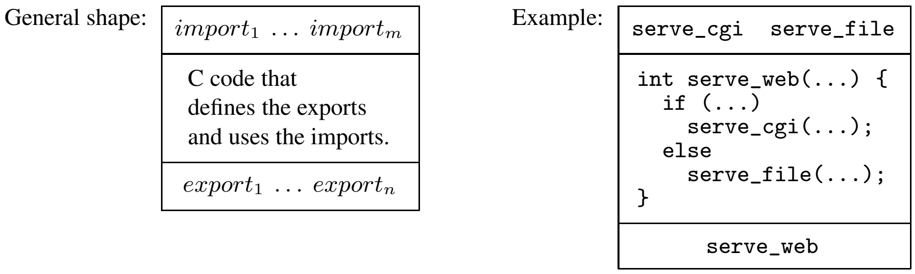


Figure 2: A unit implemented in C, ideally

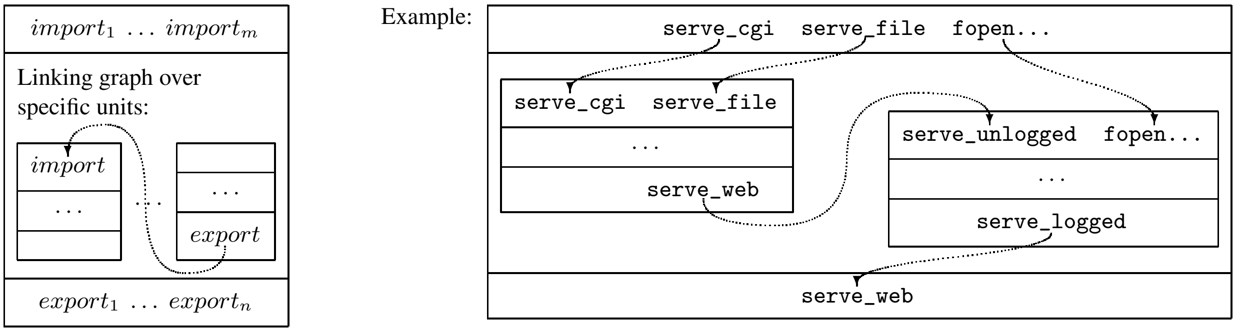


Figure 3: A compound unit, ideally

define units by referring to one or more external C files.[2] To convert the source code to compiled object files, Knit needs both the source files and their compilation flags. Figure 4 shows how the logging component's content is created by compiling `log.c` using the include directory `oskit/include`.

Third, realistic systems components have complex initialization dependencies. If there were no cyclic import relations among components, then initializations could be scheduled according to the import graph. In practice, however, cyclic imports are common, so the programmer must occasionally provide fine-grained dependency information to break cycles. A Knit unit therefore provides an explicit declaration of the unit's initialization functions, plus information about the dependencies of exports and initializers on imports. Based on these declarations, Knit automatically schedules calls to component initializers. Finalizers are treated analogously to initializers, but are called after the corresponding exports are no longer needed.

For example, the logging unit in Figure 4 defines an `open_log` function to initialize the component and a `close_log` function to finalize it. The functions exported in the `serveLog` bundle are declared to call the functions in the imported `serveWeb` and `stdio` bundles, and the initialization and finalization functions `open_log` and `close_log` rely only on the functions in the `stdio` bundle.

The `open_log` and `close_log` dependency declarations reveal a subtlety in declaring initialization constraints. The declaration "`serveLog` **needs** `stdio`" indicates that `stdio` must be initialized before any function in the bundle `serveLog` is called. However, this declaration alone does not constrain the order of initialization between the logging component and the standard I/O component; it simply says that both must be initialized before a `serveLog` function is used. In contrast, the declaration "`open_log` **needs** `stdio`" ensures that the standard I/O component is initialized before the logging component, because the logging component's initialization relies on standard I/O functions. The distinction between dependency levels is crucial to avoid overconstraining the initialization order.

A final feature needed by real units is that imports and exports may need to be renamed in order to associate Knit symbols with the identifiers used in the actual (C) implementation of a unit. For example, a serial console implementation might define a function `serial_putchar`, but export it as `putchar` to match a

[2] Knit can actually work with C, assembly, and object code. Extending Knit to handle C++, or any other language that compiles to `.o` with C-like conventions, would be straightforward but time-consuming.

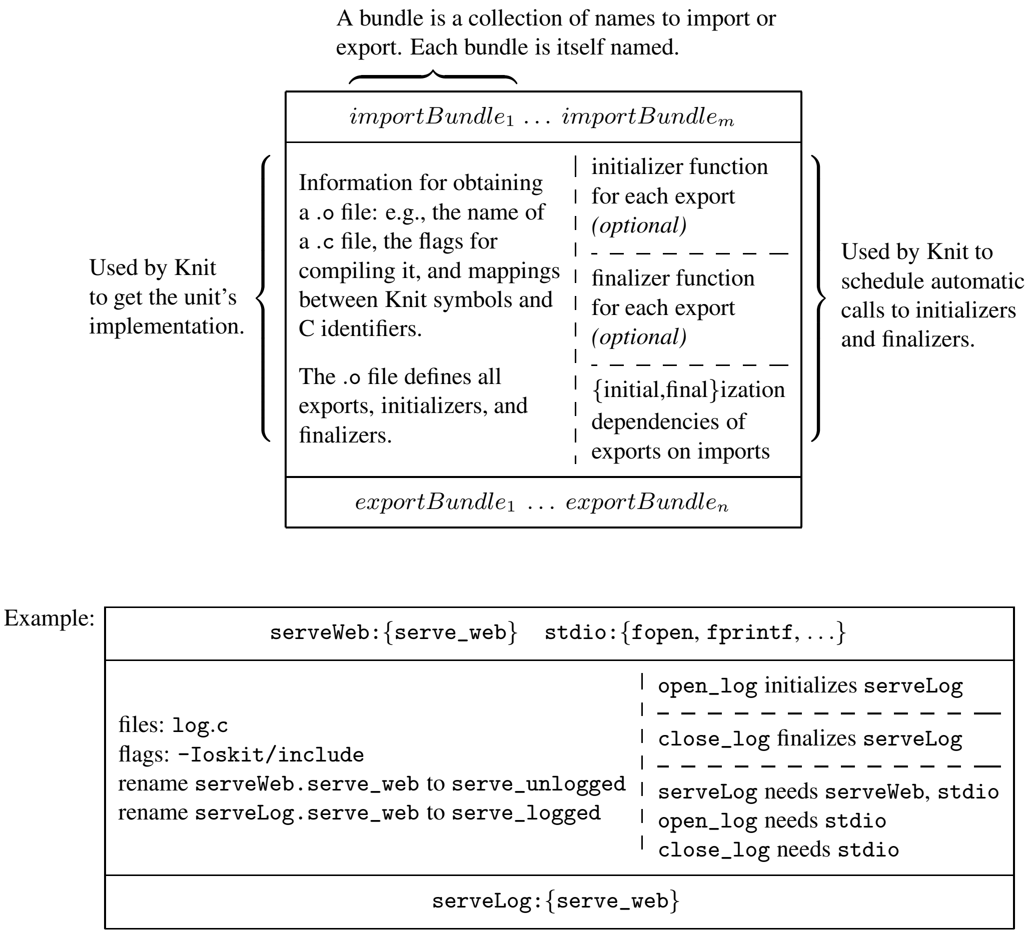


Figure 4: A unit implemented in C, more realistically. It exports a bundle `serveLog` containing the single function `serve_web`.

more generic unit interface. Another example would be a unit that both imports and exports a particular bundle type, a pattern that occurs frequently in units designed to "wrap" or interpose on other units. The logging unit shown in Figure 4 is such a unit; the implementation—the code in `log.c`—must be able to distinguish between the imported `serve_web` and the exported `serve_web` functions. This distinction is made by renaming the import or the export (or both) so that the functions have different names in the C code.

3.3 Example Code

Due to space constraints, we omit a full description of the Knit syntax. Nevertheless, to give a sense of Knit's current concrete syntax, we show how to express the running example.[3] For the sake of exposition and maintaining correspondence with the pictures, we have avoided some syntactic sugar that can shorten real unit definitions.

Figure 5 shows Knit declarations for the Web server and logging components, plus a compound unit linking them together. Before defining the units, the code defines bundle types `Serve` and `Stdio` (artificially brief in this example) and a set of compiler flags, `CFlags`. These declarations are used within the unit definitions.

As in the graphical notation in Figure 4, the `Web` unit in Figure 5 imports two functions, one for serving files and another for serving CGI scripts. The notation for both imports and exports declares a local name within the unit (the left hand side of the colon) and specifies the type of the bundle (the right hand side of the colon). This local name can be used in subsequent statements within the unit. Furthermore, all of the exports (`serveWeb`) depend on all of the imports (`serveFile` and `serveCGI`). The unit's implementation is in the file `web.c`, in Figure 6. The **rename** declarations resolve the conflict between importing and export-

[3] The syntax continues to evolve as we gain experience. Also, although we do not currently have a graphical tool for Knit, we are considering implementing one in the future.

```
bundletype Serve = { serve_web }
bundletype Stdio = { fopen, fprintf }
flags CFlags = { "-Ioskit/include" }

unit Web = {
  imports [ serveFile : Serve,
            serveCGI  : Serve ];
  exports [ serveWeb  : Serve ];
  depends {
    serveWeb needs (serveFile + serveCGI);
  };
  files { "web.c" } with flags CFlags;
  rename {
    serveFile.serve_web to serve_file;
    serveCGI.serve_web to serve_cgi;
  };
}

unit Log = {
  imports [ serveWeb : Serve,
            stdio  : Stdio ];
  exports [ serveLog : Serve ];
  initializer open_log for serveLog;
  finalizer close_log for serveLog;
  depends {
    (open_log + close_log) needs stdio;
    serveLog needs (serveWeb + stdio);
  };
  files { "log.c" } with flags CFlags;
  rename {
    serveWeb.serve_web to serve_unlogged;
    serveLog.serve_web to serve_logged;
  };
}

unit LogServe = {
  imports [ serveFile : Serve,
            serveCGI : Serve,
            stdio : Stdio ];
  exports [ serveLog : Serve ];
  link {
    [serveWeb] <- Web <- [serveFile,serveCGI];
    [serveLog] <- Log <- [serveWeb,stdio];
  };
}
```

Figure 5: Unit descriptions for parts of a Web server

web.c:

```
err_t serve_web(socket_t s, char *path) {
  if (!strncmp(path, "/cgi-bin/", 9))
    return serve_cgi(s, path + 9);
  else
    return serve_file(s, path);
}
```

log.c:

```
static FILE *log;

void open_log() {
  log = fopen("ServerLog", "a");
}

err_t serve_logged(socket_t s, char *path) {
  int r;
  r = serve_unlogged(s, path);
  fprintf(log, "%s -> %d\n", path, r);
  return r;
}
```

Figure 6: Unit internals for parts of a Web server

ing three functions with the same name by mapping the imported `serve_web` identifiers in bundles `serveFile` and `serveCGI` onto the C identifiers `serve_file` and `serve_cgi`.

The `Log` unit imports a `serve_web` function plus a bundle of I/O functions, and exports a `serve_web` function. The initialization and finalization declarations provide the same information as the graphical version of the unit in Figure 4. The dependency declarations specify that the functions `open_log` and `close_log` call functions in the `stdio` bundle, and all of the exports depend on all of the imports. Finally, the **rename** declarations resolve the conflict between the imported and exported `serve_web` functions, this time by renaming both the imported and exported versions.

The `LogServe` compound unit links the `Web` and `Log` units together, propagating imports and exports as in Figure 3. Specifically, in the **link** section,

```
[serveWeb] <- Web <- [serveFile,serveCGI]
```

instantiates a `Web` server unit using the `serveFile` and `serveCGI` imports, and binds the `Web` unit's exported bundle to the local name `serveWeb`. The next line specifies that `serveWeb` is an import, along with `stdio`, when instantiating the `Log` unit. The exported bundle of the `Log` unit is bound to `serveLog`, which is also used in the **exports** declaration of the compound unit, indicating that `Log`'s exports are propagated as exports from the compound unit.

Figure 6 shows the C code implementing the `Web` and `Log` units. Only a few details have been omitted, such as the `#include` lines. Within `web.c`, the names `serve_cgi` and `serve_file` refer to imported functions, and `serve_web` is the exported function. Similarly, in `log.c`, `serve_unlogged`, `fopen`, and `fprintf` are all imports, while `serve_logged` is an export and `open_log` is the initializer.

4 Checking Architectural Constraints

Beyond making components easier to describe and link, Knit is designed to enable powerful analysis and optimization tools for componentized systems code. In this sense, Knit serves as a bridge between low-level implementation techniques and high-level program analysis techniques.

Component kits, especially, need analysis tools to help ensure that components are assembled correctly—much more than libraries and fixed architectures need such tools. In the case of a fixed (but extensible) architecture, a programmer can learn to code by a certain set of rules and to debug each extension until it seems to follow the rules. In the case of a component kit as flexible as the OSKit, however, the rules of proper construction change depending on which components are linked together. For example, in a kernel using a "null implementation" of threads, components need not provide re-entrant procedures because the "null implementation" keeps all execution single-threaded. But when the thread component is replaced with an actual implementation of threads, the rules for proper construction of the system suddenly require re-entrant procedures.

These kinds of problems fall outside the scope of conventional checking tools such as static type systems. Type systems in most programming languages (including C, C++, etc.) express concrete properties of code, such as data representation and function calling conventions, and do not express abstract properties like deadlock avoidance or whether code is in the top half or bottom half of a device driver. More importantly, conventional type systems detect *local errors*, but the problems that occur in component software are often *global errors*, where each individual component composition may be correct, but the entire system is wrong.

To start exploring the space of possible analyses over component-based programs, we have included in Knit a simple, extensible constraint system. This system allows a programmer to define properties that Knit should check, and then lets the programmer annotate each unit declaration with the properties it satisfies.[4]

As an example property, consider the distinction between "top half" code, which includes functions like `pthread_lock` or `sleep` that require a process context, and "bottom half" code, which includes interrupt handlers that work without a context. We would like Knit to enforce the constraint that bottom-half code does not directly call top-half code, an error that might happen when a set of components is wired together incorrectly.

[4] Besides their value as checkable properties, constraints provide useful documentation of the component's behavior. Indeed, in our experience so far, constraints often duplicate information provided informally in documentation.

We can define this property in Knit with the following declarations:

```
property context
type NoContext
type ProcessContext < NoContext
```

which declare a property `context` and its two possible values, with a partial ordering on the property values that indicates `NoContext` is more general than `ProcessContext`.

Given this definition, a programmer can annotate imports and exports in units to establish property constraints. The examples below illustrate the three most common forms of annotation:

```
context(pthread_lock) <= ProcessContext
context(panic)        >= NoContext
context(printf)       <= context(putchar)
```

These three forms of constraint indicate that (1) a function (in this case, `pthread_lock`) requires a process context; (2) a function (e.g., `panic`) must work in situations where there is no process context; and (3) a function (e.g., `printf`) cannot be more flexible than some other function (e.g., `putchar`, which is used to implement `printf`). Note that the last form of constraint allows the constraints of one component to be propagated through other components in a chain of links. In practice, we find that such propagation of constraints appears most often, since most components are flexible enough to adapt to many constraint environments.

When components are linked together, Knit analyzes the components' constraints and reports an error if the constraints cannot be satisfied for some property (or if an expected constraint declaration is missing). When Knit reports a property failure, it displays the shortest chain of constraints that demonstrates the source of the problem.

5 Experience

So far, we have applied Knit to two different sets of components: (1) the OSKit [10], a large set of components that includes many legacy components, and (2) a partial implementation of the Click modular router [25], comprising a few new and cleanly constructed components.

As reported in the following sections, our experience has been positive, but with two caveats. First, the current implementation of Knit is a prototype, and the only users to date are its implementors. Second, even the implementors are unsatisfied with the current Knit syntax, which leads to linking specifications that seem excessively verbose for many tasks.

5.1 Knit and the OSKit

The OSKit is a collection of components for building operating systems. Rather than defining a fixed structure for an operating system, the OSKit provides raw materials for implementing whatever system structure a user has in mind. OSKit components can be combined in endless ways, and users are expected to write their own extensions and replacements for many kinds of components, depending on the needs of their designs. The components range in size from large, such as a TCP/IP stack derived from FreeBSD (over 18000 lines of non-blank/non-comment/etc. code), to small, such as serial console support (less than 200 lines of code). To use these components, OSKit users—many of whom know little about operating systems—must understand the interface of each component, including its functional dependencies and its initialization dependencies.

Before Knit: In the initial version of the OSKit, each component was implemented by one or more object (`.o`) files, which were stored in library archives, linked via `ld`. A component could be replaced by providing a replacement object file/library before the original library in the `ld` linking line. Since `ld` inspects its arguments in order, and since it ignores archive members that do not contribute new symbols (referenced by previously used objects), a careful ordering of `ld`'s arguments would allow a programmer to override an existing component.

As the OSKit grew in size and user base, experience soon revealed the deficiencies of `ld` as a component-linking tool. As depicted in Figure 1(c), interposition on component interfaces was difficult. Similarly, components that provided different implementations of the same interface would clash in the global namespace used for linking by `ld`. Even just checking that the linked set of components matched the intended set was difficult.

To address these issues (and, orthogonally, to represent run-time objects such as open files), a second version of the OSKit introduced COM abstractions for many kinds of components. For example, the system console, thread blocking, memory allocation, and interrupt handling are all implemented by COM components in the OSKit. For convenience, these COM objects are typically stored in a central "registry component."

Although adding COM interfaces to the OSKit solved many of the technical issues with `ld` linking, in some ways it worsened the usability problems. Programmers who had successfully used the simple function interfaces in the original OSKit at first rebelled at having to set up seemingly gratuitous objects and indirections. Programmers became responsible for getting reference counts right and for linking objects together by explicitly passing pointers among COM instances. In practice, merely getting the reference counting right was a significant barrier to experimenting with new system configurations. Furthermore, inconvenient COM interfaces proved contagious. For example, to support a kernel in which different parts of the system use different memory pools, the memory allocator component had to be made a COM object. This required changes to all code that uses allocators, changes to the code that inserts objects into the registry, and careful tweaking of the initialization order to try to ensure that objects in the registry were allocated with and subsequently used the correct allocators.

With both `ld` and COM, component linking problems interfere with the main purpose of the OSKit, which is to be a vehicle for quick experimentation. The motivation for Knit is to eliminate these problems, allowing programmers to specify which components to link together as directly as possible.

After Knit: We have converted approximately 250 components—about half of the OSKit—and about 20 example kernels to Knit. The process of developing Knit declarations for OSKit components revealed many properties and interactions among the components that a programmer would not have been able to learn from the documentation alone. Annotating a component took anywhere from 15 minutes (typically) to a full day (rarely), depending partly on the complexity of the component and its initialization requirements but mostly on the quality of the documentation (e.g., whether the imports and exports were clear).

Using Knit, we can now easily build systems that we could not build before without undue effort. For example, OSKit device drivers generate output by calling `printf`, which is also used for application output. Redirecting device driver output without Knit requires creating two separate copies of `printf`, then renaming `printf` calls in the device drivers either through cut-and-paste (a maintenance problem) or preprocessor magic (a delicate operation). Interposing on functions requires similar tricks. Such low-tech solutions work well enough for infrequent operations on a small set of names, but they do not scale to component environments in which configuration changes are frequent. Using Knit, interposition and configuration changes can be implemented and tested in just a few minutes.

Knit's automatic scheduling of initialization code was a significant aid in exploring kernel configurations. In a monolithic or fixed-framework kernel, an expert programmer can write a carefully devised function that calls all initializers in the right order, once and for all. This is not an option in the OSKit, where the correct order depends on which components are glued together. Previous versions of the OSKit provided canned initialization sequences, but, as just described, using these sequences would limit the programmer's control over the components used in the configuration. Knit allows the expert to annotate components with their dependencies and allows client programmers to combine precisely the components they want with reliable initialization. Annotations for device drivers, filesystems, networking, console, and other intertwined components have proven relatively easy to get right at the local level, and the scheduler has performed remarkably well in practice.

The constraint system described in Section 4 caught a few small errors in existing OSKit kernels, written by ourselves, OSKit experts. We added constraints to kernels composed of roughly 100 units. Among those units, 35 required the addition of constraints, of which 70% simply propagated their context from imports to exports using the constraint "`context(exports) <= context(imports)`" or stated that a component could be used without a process context. These required little effort. The remainder (device drivers and thread packages) required more care because we had to examine the source code to determine how individual components were used. The errors we found were easy to fix once identified. The advantage of Knit is that its constraint system found the bugs, and will continue to detect new bugs as the code evolves.

A further benefit of using Knit is that it makes it easier to create small, special-purpose kernels. The combination of knowing exactly which components are in our kernels (and why) and the ease of replacing one component with another enabled us to dramatically reduce the size of some kernels. An extreme example is our smallest kernel (the toy `hello_world` kernel) which is four times smaller when built with Knit than without.

The Knit version of the OSKit continues to use COM for subsystems that behave more like objects than modules. For example, individual files and directories are still implemented as COM objects.

5.2 Clack

The elegant Click modular router [25] allows a programmer, or even a network administrator, to build a special-purpose router by wiring together a set of components. Click provides its own language for configuring routers, so that a programmer might write

```
FromDevice(eth) -> Counter -> Discard
```

to create a "router" that counts packets.

Click is implemented in C++, and each router component is implemented by a C++ class instance. A programmer can add new kinds of router components to Click by deriving new C++ classes. To demonstrate that Knit is general and more than just a tool for the OSKit, we implemented a subset of Click version 1.0.1 with Knit components instead of C++ classes.[5] We dubbed our new component suite *Clack*.

[5] Because Click's router components are generally very small and functionally simple, much of the actual component source code deals with the Click-specific component framework and not with the functional purpose of the components. For this reason, we decided to write our components from scratch rather than adapt the existing Click components to Knit.

Given Click as a model, implementing enough of Clack in Knit to build an IP router (without handling fragmentation or IP options) took a few days. A typical Clack component required several lines of C plus several lines of unit description. Clack follows the basic architecture of Click, but the details have been Knitified. For example, Click supports component initialization through user-provided strings. Clack emulates this feature with trivial components that provide initialization data. Similarly, Click's support for a (configure-time) variable number of imports or exports is handled in Clack with appropriate fan-in and fan-out components. Clack does not emulate the more dynamic aspects of Click, such as allowing a component to locate certain other components at run time.

Overall, by avoiding the syntactic overhead required to retrofit C++ classes as components, Clack definitions are considerably more compact than corresponding Click definitions (by roughly a factor of three for small components). The size of Clack `.o` files was smaller than Click `.o`'s by an even more dramatic amount (roughly a factor of seven for small components). This is mainly due to Clack's fine-grained control of the router's content and Click's support for dynamic composition. The overall performance of Clack is comparable to that of Click.

In contrast, using the full Knit linking language to join Clack components is more complex than using Click's special-purpose language. If Clack were to be used by network administrators, we would certainly build a (straightforward) translator from Click linking specifications to Knit linking expressions.

Based on our small experiment, we believe that Knit would have been a useful tool for implementing the original Click component set. The Click architecture fits well in the Knit language model, and the Click configuration language is conceptually close to the Knit linking model. The one aspect of Click that does not fit well into Knit is the rapid deployment of new configurations. Click configurations consist of C++ object graphs that can be dynamically generated, whereas Clack configurations are resolved at link time. Note, however, that recent work on Click performance by its authors also conflicts with dynamic configuration [19].

To the extent that Knit is a bridge to analyses and optimizations, we believe that Knit would be a superior implementation environment for Click compared to C++. In Section 6, we report on cross-component optimizations in Knit, and we show that they substantially increase the performance of Clack. The constraint-checking facilities of Knit can also be used to enforce configuration restrictions among Clack components, ensuring, for example, that components only receive packets of an appropriate type (Ethernet, IP, TCP, ARP, etc.).

These analyses are only a start, and a Knit-based Click would be able to exploit future Knit developments.

6 Implementation and Performance

Component software tends to have worse performance than monolithic software. Introducing component boundaries invariably increases the number of function calls in a program and hides opportunities for optimization. However, Knit's static linking language allows it to eliminate these costs. Indeed, we can achieve useful levels of optimization while exploiting the existing infrastructure of compilers and linkers.

In a typical use, the Knit compiler reads the linking specification and unit files, generates initialization and finalization code, runs the C compiler or assembler when necessary, and ultimately produces object files. The object files are then processed by a slightly modified version of GNU's `objcopy`, which handles renaming symbols and duplicating object code for multiply-instantiated units. Finally, these object files are linked together using `ld` to produce the program.

To verify that Knit does not impose an unacceptable overhead on programs, we timed Knit-based OSKit programs that were designed to spend most of their time traversing unit boundaries. We compared these programs with equivalent OSKit programs built using traditional tools. The number of units in the critical path ranged between 3 and 8 (including units such as memory file systems, VGA device drivers, and memory allocators), with the total number of units between 37 and 72. Tests were run between 10 and a million times, as appropriate. Knit was from 2% slower to 3% faster, $\pm 0.25\%$. Note that these experiments were done without applying the optimization that we describe next.

For cross-component optimization, we have implemented a strategy that is deceptively simple to describe: Knit merges the code from many different C files into a single file, and then invokes the C compiler on the resulting file. The task of merging C code is simple but tedious; Knit must rename variables to eliminate conflicts, eliminate duplicate declarations for variables and types, and sort function definitions so that the definition of each function comes before as many uses as possible (to encourage inlining in the C compiler). Fortunately, these complexities are minor compared to building an optimizing compiler. To limit the size of the file provided to the compiler, Knit can merge files at any unit boundary, as directed by the programmer via the unit specifications. When used in conjunction with the GNU C compiler (which has poor interprocedural optimization), this enables functions to be inlined across component boundaries which may, in turn, enable further intraprocedural optimizations such as constant folding and common subexpression elimination.[6]

To test the effectiveness of Knit's optimization technique (which we call *flattening*), we applied it to our Clack IP router. Since our focus was on the structure of the router, we flattened only the router rather than the entire kernel. For comparison, we rewrote our router components in a less modular way: combining 24 separate components into just 2 components, converting the result to idiomatic C, and eliminating redundant data fetches. The most important measure of an optimization is, of course, the time the optimized program takes. In this case, we also measured the impact of stalls in the instruction fetch unit because there is a risk that the inlining enabled by flattening would increase the size of the router code, leading to poor I-cache performance.[7] Our experiments were performed on three 200 MHz Pentium Pro machines, each with 64 MB of RAM and 256 KB of L2 cache, directly connected via DEC Tulip 10/100 Ethernet cards, with the "machine in the middle" functioning as the IP router.

The results are shown in Table 1. The manual transformation gives a significant (21%) performance improvement, demonstrating that componentization can have significant overhead. Flattening the modular version of the router gives an even more significant (35%) improvement: rather than harming I-cache behavior, flattening greatly improves I-cache behavior. Examination of the assembly code reveals that flattening eliminates function call overhead (e.g., the cost of pushing arguments onto the stack), turns function call nests into compact straight-line code, and eliminates redundant reads via common subexpression elimination. Combining both optimizations gives only a small (5%) additional improvement in performance, suggesting that the optimizations obtain their gains from the same source. Our overall conclusion is that we can eliminate most of the cost of componentization by blindly merging code, enabling conventional optimizing compilers to do the rest.

Meanwhile, the authors of Click have been working on special-purpose optimizations for their system [19]. Their optimizations include a "fast classifier" that generates specialized versions of generic components, a "specializer" that makes indirect function calls direct, and an "xform" step that recognizes certain patterns of components and replaces them with faster ones. While their code base and optimizations are very different from ours, the relative performance of their system and the effectiveness of their optimizations provides a convenient touchstone for our results. The performance of their base

[6] We used `gcc` version 2.95.2 for all our experiments.

[7] We also measured number of instruction misses in the L1 and L2 caches: both the overall downward trend and the approximate ratio between the three numbers were the same across all experiments.

hand optimized	flattened	cycles	instr. fetch stall cycles	text size (bytes)
		2411	781	109464
√		1897	637	108246
	√	1574	455	106065
√	√	1457	361	106305

Table 1: Clack router performance using various optimizations, measured in number of cycles from the moment a packet enters the router graph to the moment it leaves. I-fetch stalls were measured using the Pentium Pro counters and are reported in cycles. The i-fetch stall numbers along with the given code sizes reveal that inlining did not have a negative effect.

version	cycles
unoptimized	2486
optimized	1146

Table 2: Click router performance, with and without all three MIT optimizations, measured as above. The Click routers were executed in the same OSKit-derived kernel and on the same hardware as the Clack routers.

and optimized systems is shown in Table 2. We note that the performance of their base system is approximately the same as ours (3% slower) but that the effect of applying all three Click optimizations is significantly better than the two Clack optimizations (54%). Considering that Knit achieves its performance increase by blindly merging code, without any profiling or tuning of Clack by programmers, we again interpret the results of our experiment to indicate that Knit would make a good implementation platform for Click-like systems. We believe that Knit would save implementors of such systems time and energy implementing basic optimizations, allowing them to concentrate on implementing domain-specific or application-specific optimizations.

The core of our current Knit compiler prototype consists of about 6000 lines of Haskell code, of which roughly 500 lines implement initializers and finalizers, 500 lines implement constraints, and 1500 lines implement flattening. Our prototype implementation is acceptably fast—more than 95% of build time is spent in the C compiler and linker—although constraint-checking more than doubles the time taken to run Knit.

7 Related Work

Much of the early research in component-based systems software involves the design and implementation of microkernels such as Mach [1] and Spring [13]. With an emphasis on robustness and architectures for flexibility and extensibility at subsystem boundaries, microkernel research is essentially complementary to research on component implementation and composition tools like Knit. More recently, the Pebble [11] microkernel-based OS has an emphasis on flexibly combining components in protection domains, e.g., in separate servers or in a single protection domain. However, the actual set of components is quite fixed.

More closely related research involves the use of component kits for building systems software. MMLite [16] followed our OSKit lead by providing a variety of COM-based components for building low-level systems. MMLite takes a very aggressive approach to componentization and provides certain features that the current OSKit and Knit lack, such as the ability to replace system components at run-time. The Scout operating system [24] and its antecedent *x*-kernel [17] consist of a modest number of modules that can be combined to create software for "network appliances" and protocols. The Click system [25] also focuses on networking, but specifically targets packet routers (e.g., IP routers) and its components are much smaller than Scout's. Scout and Click, like Knit, rely on "little languages" outside of C to build and optimize component compositions, and to schedule component initializations, but only Knit provides a general-purpose language that can be used to describe both new and existing components. Languages like C++ and Java have also dealt with automatic initialization of static variables, but through complex (and, in the case of C++, unpredictable) rules that give the programmer little control.

Knit's ability to work with unmodified C code distinguishes it from projects such as Fox [15] and Ensemble [20], which rely on a high-level implementation language, or systems such as pSOS [31] and the currently very small eCos [29]. Both eCos and pSOS provide configuration languages/interfaces but neither really has "components": individual subsystems can be included or excluded from the system, but there is no way to change the interconnections between components. In contrast to Ensemble, it should be noted that Knit provides a more "lightweight" system for reasoning about component compositions. This is a deliberate choice: we intend for Knit to be usable by systems programmers without training in formal methods.

Like Knit, OMOS [27,28] enables the reuse of existing code through interface conversion on object files (e.g., renaming a symbol, or wrapping a set of functions). Unlike Knit, however, OMOS does not provide a configuration language that is conducive to static analysis.

Commercial tools, such as Visual Basic and tools using it, help programmers design, browse, and link software components that conform to the COM or CORBA standards. Such tools and frameworks currently lack the kinds of specification information that would enable automated checking of component configurations beyond datatype interfaces, and the underlying object-based model of components makes cross-component optimization exceedingly difficult.

Knit's unit language is derived from the component model of Flatt and Felleisen [9], who provide an extensive overview of related programming languages research. Module work that subsequently achieved similar goals includes the recursive functors of Crary et al. [5] and the typed-assembly linker of Glew and Morrisett [12].

CM [3] solves for ML many of the same problems that Knit solves for C, but ML provides CM with a pre-existing module language and a core language with well-defined semantics. Unlike Knit, CM disallows recursive modules, thus sidestepping initialization issues. Further, CM relies on ML's type system to perform consistency checks, instead of providing its own constraint system.

The lazy functional programming community routinely uses higher-order functions to glue small components into larger components, relies on sophisticated type systems to detect errors in component composition, and makes use of lazy evaluation to dynamically determine initialization order in cyclic component graphs. For example, the Fudgets GUI component library [4] uses a dataflow model similar to the one used in Click (and Clack) but the data sent between elements can have a variety of types (e.g., menu selections, button clicks, etc.) instead of a single type (e.g., packets). This approach has been refined and applied to different domains by a variety of authors.

The GenVoca model of software components [2] has several similarities to our work: their "realms" correspond to our bundle types, their "type equations" correspond to our linking graphs, and their "design rules" correspond to our constraint systems. In its details, however, the GenVoca approach is quite different from ours. GenVoca is based on the notion of a *generator*, which is a compiler for a domain-specific language. GenVoca components are program transformations rather than containers of code; a GenVoca compiler therefore synthesizes code from a high-level (and domain-specific) program description. In contrast, Knit promotes the reuse of existing (C) code and enables flexible composition through its separate, unit-based linking language. Notably, Knit allows cyclic component connections—important in many systems—and can check constraints in such graphs. The GenVoca model of components and design rules, however, is based on (non-cyclic) component trees.

8 Conclusion

From our experiences in building and using OSKit components, and that of our clients in using them, we believe that existing tools do not adequately address the needs of componentized software. To fill the gap, we have developed Knit, a language for defining and linking systems components. Our initial experiments in applying Knit to the OSKit show that Knit provides improved support for component programming.

The Knit language continues to evolve, and future work will focus on making components and linking specifications easier to define. In particular, we plan to generalize the constraint-checking mechanism to reduce repetition between different constraints and, we hope, to unify scheduling of initializers with constraint checking. We may also explore support for dynamic linking, where the main challenge involves the handling of constraint specifications at dynamic boundaries. Continued exploration of Knit within the OSKit will likely produce improvements to the language and increase our understanding of how systems components should be structured.

Knit is a first step in a larger research program to bring strong analysis and optimization techniques to bear on componentized systems software. We expect such tools to help detect deadlocks, detect unsafe locking, reduce abstraction overheads, flatten layered implementations, and more. All of these tasks require the well-defined component boundaries and static linking information provided by Knit. We believe that other researchers and programmers who are working on componentized systems could similarly benefit by using Knit.

Availability

Source and documentation for our Knit prototype is available under `http://www.cs.utah.edu/flux/`.

Acknowledgments

We are grateful to Mike Hibler, Patrick Tullmann, John Regehr, Robert Grimm, and David Andersen for providing many comments and suggestions that helped us to improve this paper. Mike deserves special thanks, for special help. We are indebted to the members of the MIT Click group, both for Click itself and for sharing their work on Click optimization long before that work was published. Finally, we thank the anonymous reviewers and our shepherd, David Presotto, for their constructive comments and suggestions.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.

[2] D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, pages 67–82, Feb. 1997.

[3] M. Blume and A. W. Appel. Hierarchial Modularity. *ACM Transactions on Programming Languages and Systems*, pages 812–846, July 1999.

[4] M. Carlsson and T. Hallgren. Fudgets: A Graphical User Interface in a Lazy Functional Language. In *Proc. of the Conference on Functional Programming and Computer Architecture*, 1993.

[5] K. Crary, R. Harper, and S. Puri. What is a Recursive Module? In *Proc. ACM SIGPLAN '99 Conf. on Programming Language Design and Implementation (PLDI)*, pages 50–63, Atlanta, GA, May 1999.

[6] S. Dorward and R. Pike. Programming in Limbo. In *Proc. of the IEEE Compcon 97 Conf.*, pages 245–250, San Jose, CA, 1997.

[7] R. B. Findler and M. Flatt. Modular Object-Oriented Programming with Units and Mixins. In *Proc. of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 94–104, Baltimore, MD, Sept. 1998.

[8] M. Flatt. *Programming Languages for Component Software*. PhD thesis, Rice University, June 1999.

[9] M. Flatt and M. Felleisen. Units: Cool Units for HOT Languages. In *Proc. ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation (PLDI)*, pages 236–248, June 1998.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.

[11] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In *Proc. of the USENIX 1999 Annual Technical Conf.*, June 1999.

[12] N. Glew and G. Morrisett. Type-Safe Linking and Modular Assembly Language. In *Proc. of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, TX, Jan. 1999.

[13] G. Hamilton and P. Kougiouris. The Spring Nucleus: a Microkernel for Objects. In *Proc. of the Summer 1993 USENIX Conf.*, pages 147–159, Cincinnati, OH, June 1993.

[14] S. P. Harbison. *Modula-3*. Prentice Hall, 1991.

[15] B. Harper, E. Cooper, and P. Lee. The Fox Project: Advanced Development of Systems Software. Computer Science Department Technical Report 91–187, Carnegie Mellon University, 1991.

[16] J. Helander and A. Forin. MMLite: A Highly Componentized System Architecture. In *Proc. of the Eighth ACM SIGOPS European Workshop*, pages 96–103, Sintra, Portugal, Sept. 1998.

[17] N. C. Hutchinson and L. L. Peterson. Design of the x-kernel. In *Proc. of SIGCOMM '88*, pages 65–75, Aug. 1988.

[18] International Organization for Standardization. *Ada 95 Reference Manual. The Language. The Standard Libraries*, Jan. 1995.

[19] E. Kohler, B. Chen, M. F. Kaashoek, R. Morris, and M. Poletto. Programming language techniques for modular router configurations. Technical Report MIT–LCS–TR–812, MIT Laboratory for Computer Science, Aug. 2000.

[20] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 80–92, Dec. 1999.

[21] D. MacQueen. Modules for Standard ML. In *Proc. of the 1984 ACM Conf. on Lisp and Functional Programming*, pages 198–207, 1984.

[22] Microsoft Corporation and Digital Equipment Corporation. *Component Object Model Specification*, Oct. 1995. 274 pages.

[23] J. G. Mitchell, W. Mayberry, and R. Sweet. *Mesa Language Manual*, 1979.

[24] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A Communications-oriented Operating System. Technical Report 94–20, University of Arizona, Dept. of Computer Science, June 1994.

[25] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 217–231, Dec. 1999.

[26] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, June 1999. Revision 2.3. OMG document formal/98–12–01. Part of the CORBA 2.3 specification.

[27] D. B. Orr, J. Bonn, J. Lepreau, and R. Mecklenburg. Fast and Flexible Shared Libraries. In *Proc. of the Summer 1993 USENIX Conf.*, pages 237–251, June 1993.

[28] D. B. Orr, R. W. Mecklenburg, P. J. Hoogenboom, and J. Lepreau. Dynamic Program Monitoring and Transformation Using the OMOS Object Server. In D. Lilja and P. Bird, editors, *The Interaction of Compilation Technology and Computer Architecture*. Kluwer Academic Publishers, 1994.

[29] Red Hat, Inc. eCos: Embedded Configurable Operating System, Version 1.3. `http://www.redhat.com/products/ecos/`.

[30] C. A. Szyperski. Import Is Not Inheritance — Why We Need Both: Modules and Classes. In *ECOOP '92: European Conf. on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, July 1992.

[31] WindRiver, Inc. pSOS. `http://www.windriver.com/`.