SOFTWARE VARIABILITY MECHANISMS FOR IMPROVING RUN-TIME PERFORMANCE

by

Eric Norman Eide

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2012

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

The dissertation of Eric Norman Eide

has been approved by the following supervisory committee members:

Matthew Flatt	, Chair	October 2, 2012 Date Approved
Robert R. Kessler	, Member	October 2, 2012 Date Approved
Gary Lindstrom	, Member	October 2, 2012 Date Approved
John Regehr	, Member	October 2, 2012 Date Approved
Shriram Krishnamurthi	, Member	October 15, 2012 Date Approved

and by Alan Davis, Chair of the School of Computing

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

A variability mechanism is a software implementation technique that realizes a choice in the features that are incorporated into a software system. Variability mechanisms are essential for the implementation of configurable software, but the nature of mechanisms as structuring abstractions is not well understood. Mechanisms are often equated with their stereotypical realizations. As a consequence, certain variability mechanisms are generally viewed as having limited applicability due to run-time performance overhead.

We claim that it is feasible and useful to realize variability mechanisms in novel ways to improve the run-time performance of software systems. This claim is supported by the implementation and evaluation of three examples.

The first is the *flexible generation of performance-optimized code from high-level specifications*. As exemplified by Flick, an interface definition language (IDL) compiler kit, concepts from traditional programming language compilers can be applied to bring both flexibility and optimization to the domain of IDL compilation.

The second is a method for *realizing design patterns* within software that is neither object-oriented nor, for the most part, dynamically configured. By separating static software design from dynamic system behavior, this technique enables more effective and domain-specific detection of design errors, prediction of run-time behavior, and more effective optimization. The method is demonstrated using a suite of operating-system components.

The third, *middleware-based brokering of resources*, can improve the ability of multi-agent, real-time applications to maintain quality of service (QoS) in the face of resource contention. A CPU broker, for instance, may use application feedback and other inputs to adjust CPU allocations at run time. This helps to ensure that applications continue to function, or at least degrade gracefully, under load.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

Chapter 2 is a reprint of “Flick: A Flexible, Optimizing IDL Compiler,” a published work by the author of this dissertation and his colleagues:

Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. “Flick: A Flexible, Optimizing IDL Compiler.” In *Proceedings of the ACM SIGPLAN ’97 Conference on Programming Language Design and Implementation (PLDI)*, pages 44–56, June 1997. © 1997 Association for Computing Machinery, Inc. Reprinted by permission. `http://doi.acm.org/10.1145/258915.258921`

The work in Chapter 2 was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602–96–2–0269.

Chapter 3 is a reprint of “Static and Dynamic Structure in Design Patterns,” a published work by the author of this dissertation and his colleagues:

Eric Eide, Alastair Reid, John Regehr, and Jay Lepreau. “Static and Dynamic Structure in Design Patterns.” In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 208–218, May 2002. © 2002 Association for Computing Machinery, Inc. Reprinted by permission. `http://doi.acm.org/10.1145/581339.581367`

The work in Chapter 3 was largely supported by the Defense Advanced Research Projects Agency, monitored by the Air Force Research Laboratory, under agreements F33615–00–C–1696 and F30602–99–1–0503.

Chapter 4 is a reprint of “Dynamic CPU Management for Real-Time, Middleware-Based Systems,” a published work by the author of this dissertation and his colleagues. The following notice applies to Chapter 4:

© IEEE, 2004. Reprinted, with permission, from *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 286–295, May 2004: “Dynamic CPU Management for Real-Time, Middleware-

Based Systems” by Eric Eide, Tim Stack, John Regehr, and Jay Lepreau. `http://dx.doi.org/10.1109/RTTAS.2004.1317274`

The work in Chapter 4 was largely supported by the Defense Advanced Research Projects Agency, monitored by the Air Force Research Laboratory, under agreement F33615–00–C–1696.

CHAPTER 1

INTRODUCTION

This dissertation provides three examples of novel software variability mechanisms to support the following hypothesis:

It is feasible and useful to realize variability mechanisms in novel ways to improve the run-time performance of software systems.

Each of the examples is based on a well-known style of modular software composition, and each modifies the composition mechanism while maintaining implementation modularity. Each example shows that, by appropriately modifying the technique used to compose software "parts," the run-time performance of software systems can be improved. Composed systems can be made faster or more predictable.

1.1 Software Configuration and Variation

Because software systems are complex, a program today is essentially never made from just a single part. Instead, any nontrivial program is a composition of many pieces. The exact composition is the result of many individual decisions, made over a potentially long period of time by many different people. The chain of decisions starts even before the software is implemented—i.e., during the requirements analysis phase of the software life cycle—and may continue even after the program has started to execute.

The *architecture* of a software system describes how the parts of the system fit together. An *architectural style* defines what it means to be a "part" and what it means for parts to be connected to one another. For example, the architecture of a software system might be described as a collection of files that are stored and organized within a file system. A second architectural style might describe software systems as compositions of module or class definitions, written in a particular language and connected to one another according to the rules of the language. A third style might describe software structures in terms of objects that are created and connected during the execution of a process. A fourth might describe a (distributed) system as a collection of processes: the processes execute over a set of one or more physical computers and exchange messages with one another via a network.

An architectural style provides a language—often nontextual, and often informal—for constructing models of software systems. More generally, an architectural model does not describe just a single instance of a software system. Instead, a typical model describes a family of systems. The members of the family share an architectural "blueprint" but differ in their details, because the blueprint or its associated software-development process defines points at which decisions must be made.

A model of a distributed system, for example, may not define the network locations (e.g., IP addresses) of the computers on which the distributed processes execute. Omitting this detail from the model allows the model to be more general; the model thereby can describe not just a particular instance of a system, but instead all instances, independently of the locations of processes. The processes' locations are parameters of the systems that are described by the model.

A more interesting kind of decision occurs when a software developer, installer, or user decides how a software system should be composed from parts. In contrast to the example above, which concerns the implementation of a connection within a fixed graph of communicating parts (processes), architectural decisions can also involve choices among the parts themselves and their assembly. For example, consider a software system that requires persistent storage. The designer of this system might decide that the system should allow persistent data to be stored either in a file system or in a database—a choice between two software components that each implement persistent storage. The system's programmer may then decide to implement the choice as a compile-time option. The installer then chooses between the options when he or she compiles the software.

This example illustrates that a software designer must make certain choices about parts and their composition when he or she creates an initial architecture. However, a designer can also choose to defer certain decisions. Doing so makes it possible for a product assembler, system administrator, or user to decide the exact set of parts that will be combined within an instance of the software system. In the current example, at design time, a person established an option and deferred the decision-making for that option. At implementation time, another person coded the option in a way that further deferred the decision-making. One can imagine that at compile time, the system installer could further defer the choice for the persistent storage mechanism, perhaps by selecting a software module that implements the choice at run time.

At the level of architectural descriptions, *feature models* are popular for describing the choices that are defined within the description of a software family (often called a *software product line* [5]). A feature model highlights the places where configuration decisions must be made, e.g., where one must choose to include or exclude features from a software configuration. The complete implementation of a software product line, as described by a feature model, includes not only the implementations of the described software features but also mechanisms that support the configuration options that are described by the model. Many software implementation techniques—including modules [25, page 122], components [30], design patterns [13], frameworks [17], plug-ins [31, page 553], dependency injection [11], and others—can be used in isolation or combination to implement decisions about software assembly.

This dissertation is concerned with how the decision points defined by software architectures are implemented, and how the decisions are ultimately made.

1.2 Variability Mechanisms

A *variability mechanism* is a software implementation technique that realizes a choice in the features that are incorporated into a software system [16]. This definition is intentionally broad. It refers not only to the use of specific source-code language constructs—e.g., virtual functions—but also to idioms that use source-code constructs in combination to create more complex structures, such as design patterns. The notion of a variability mechanism also includes the realization of choices outside of the basic source code of a system. In particular, things such as build scripts (e.g., Makefiles [19]), preprocessing scripts (e.g., Autoconf [33]), compiler options, and static and dynamic linking can be used to realize choices in the ways that software is composed, configured, and deployed. Such a broad view is useful for addressing a wide range of software features. In the context of describing variability mechanisms, we take "feature" to mean any software functionality or property that a software architect or designer wishes to treat as a unit of composition or configuration.

A wide variety of variability mechanisms have been cataloged and studied [2, 12], and they have been classified according to numerous qualities [29]. The most inherent qualities of any variability mechanism, however, arise from its binding time and its binding sites.

1.2.1 Binding Time

The *binding time* of a variability mechanism is the point in the software life cycle at which an instance of the mechanism may be utilized to make a composition or configuration decision. The binding time of a mechanism is typically described as one of the following:

- *Compile time:* the time at which the source code of the software system is translated into machine code, byte code, or another internal form.
- *Link time:* the time when the compiled units of a system are composed to form a complete program. This includes composition with compiled units that are provided by the program's execution environment, e.g., system libraries.
- *Initialization time:* the time between the invocation of the program and the time at which the program enters its normal mode of operation. It is during this time that a program creates its initial data structures, reads configuration files and command-line options, and performs other preparatory actions.
- *Run time:* the time during the normal execution of the program, following its initialization phase.

Finer-grain distinctions are possible [6, 12], and the phases listed above may be overlapping or even recurring. Although such distinctions are possible, for the purposes of this dissertation, it is sufficient to classify variability mechanisms using the basic four times listed above. These are sufficient to describe the effects of binding time on one of the primary concerns addressed in this dissertation: the run-time performance of software.

In general, mechanisms that support compile-time binding are associated with improved run-time performance of the final, configured software. When a feature selection is known at compile time, the feature's interactions with the rest of the software system can be more easily analyzed by the system's compiler. Excluded features also create opportunities for analysis, through their known absence. These analyses can enable well-known, significant optimizations, which in turn can improve the ultimate run-time performance of the compiled software.

The cost of compile-time binding comes in terms of subsequent flexibility of the compiled software. When a binding decision is made at compile time, it cannot generally be unmade at a later point in the software life cycle. This inflexibility may be intended by the provider of the software, but it may nevertheless be a hindrance to downstream uses of the compiled software by users and software integrators.

In contrast to variability mechanisms with compile-time binding, mechanisms with initialization-time or run-time binding are generally associated with configuration decisions that are expected to be made by site administrators or users. (Dynamic linking supports such decisions as well.) Such binding times allow a single program to be more generally useful to consumers: for example, the parsing of command-line options allows a user to enable and disable features each time he or she executes the software. Run-time binding can be a building block for other capabilities such as software updates and user-selected software plug-ins.

In comparison to a variability mechanism that has compile-time binding, a mechanism with initialization-time or run-time binding may introduce performance overhead into the software. This overhead stems from two factors. The first is the cost associated with the mechanism itself: this includes time spent parsing feature-selection inputs (e.g., command-line options), time spent configuring selected features, and time spent evaluating the language constructs that implement the connections to selected features (e.g., testing variables, or dispatching virtual function calls). The second source of overhead, in comparison to compile-time mechanisms, is due to the compile-time optimization opportunities that are lost when run-time binding mechanisms are used. These sources of overhead can be significant for some applications, such as the implementation of network communication protocols [20, 21].

Initialization-time and run-time binding do not always result in run-time performance overheads, of course. If the dynamically selected features perform better than the features that would have been selected statically, then dynamic feature selection can lead to overall performance improvements for the software system as a whole. Chapter 4 details an example of such a system.

1.2.2 Binding Sites

In the language of software product lines, a *binding site* is a point within the realization of a software product at which a variability mechanism is applied to implement a variation point [6]. A single variation point in a conceptual feature model may correspond to numerous binding sites within the implementation of a software product. Collectively, the binding sites that make up the implementation of a variation point are an instance of the variability mechanism.

The *modularity* of a variability mechanism refers to the ability to isolate typical instances of the mechanism within a well-defined—and ideally small—set of product implementation constructs.[1] This may be isolation at the level of source code (e.g., files and function definitions) or it may be isolation within non-source-code artifacts such as build and deployment scripts. Two factors influence the isolation of binding sites, and thus the degree of modularity associated with a variability mechanism.

The first is the degree to which binding sites must be implemented individually as hand-coded changes to the implementation. If a particular variation point requires many binding sites, and if each binding site is implemented as a hand-coded change to the software source, then the instances of the mechanism will be *scattered* over the many parts of the software [9, page 4]. Scattering can make it difficult to identify all of the code that implements a variation point (i.e., all of the binding sites); scattering can also make it difficult to change the instance of the variability mechanism.

The second factor is the degree to which binding sites for a variation point must be located with code or other implementation constructs that stem from concerns other than the implementation of the variation point. When a binding site must be inserted into code that exists for other concerns, it may be *tangled* with that other code [9, page 4]. Tangling refers to situations in which the implementations of multiple concerns become intermixed within a single programming artifact, e.g., a function, class, or a source file. Variability mechanisms that require such tangling harm modularity by causing programming artifacts to become associated with more than one concern—and often, by not clearly delineating which parts of an artifact belong to which concerns.

Consider, for example, the use of dynamic linking to choose among possible implementations of a library. On modern operating systems, dynamic linking is often implemented so as to be transparent to a system's source code—thus, it has no binding sites in the source code at all (and thus no source-level scattering or tangling). Rather than requiring changes to source code, the primary requirement for using dynamic linking is to write a system's build scripts so that they collect the system's compiled units into separate libraries as desired. Given that dynamic linking implements a variation point, and a variation point is by definition a point of connection between features, the library boundaries in this case follow the feature boundaries. A programmer would likely implement a compilation-unit

[1]Particular mechanisms may be modular in other respects as well. For example, a feature implemented as a class in an object-oriented language may be described as modular in the sense that it hides the details of its implementation from its callers. This form of modularity also provides software-engineering benefits, but it is not the intended meaning of modularity in this section.

boundary between feature implementations in any case, regardless of how the systems' features are ultimately linked. Thus, dynamic linking is an example of a variability mechanism that often exhibits high modularity—the implementation of the mechanism often involves few changes to the implementation of a software system overall.

One can contrast the qualities of dynamic linking with the qualities of conditional compilation, which is another commonly used variability mechanism. Conditional compilation uses a preprocessing tool, such as the C preprocessor, to manipulate the source code of a system to include or exclude code that is associated with given features. Typically, each selectable feature is associated with a unique preprocessor symbol. When the preprocessor is executed, the values of the features' symbols are set so that the implementation of each feature is included in or excluded from the output of the preprocessor, according to the wishes of the person configuring the software.

In contrast to dynamic linking, conditional compilation often involves many binding sites within the implementation of a software system. Each place that the feature's preprocessor symbol is set or checked is a binding site.[2] The number of such sites can be small or large depending on the requirements of the feature being controlled. In contrast to dynamic linking, each check of a preprocessor symbol involves a change to the source code of the software—i.e., potentially many individual changes spread across multiple program artifacts [18, 26], thus leading to scattering. Because the preprocessor directives are "inlined" into the implementation of other concerns, tangling can be a problem as well. Indeed, if numerous features are controlled via conditional compilation, all of the binding sites—the preprocessor directives and segments of code controlled by those directives—can impair the readability of the source code as a whole [27]. The binding sites for different features may also interfere with each other, leading to complicated preprocessing directives. Such heavy and convoluted use of the C preprocessor for software configuration is commonly known as "#ifdef hell" [18]. All of these qualities suggest that conditional compilation is not a very modular variability mechanism in general.

[2]As we have described conditional compilation, the symbols associated with features are used only by the preprocessor; their values are not incorporated into the output of the preprocessor. We do not discuss other ways to use a preprocessor to implement variability mechanisms here. Ernst et al. classify common uses of preprocessor macros in C [8]; many of the idioms they identify can be used to implement feature binding sites.

1.3 Selecting a Variability Mechanism

In choosing the implementation technique for a variation point, a software designer or programmer must consider several issues. These arise from the requirements of the variation point. For example, to decide if a particular variability mechanism is suitable for implementing a particular variation point, a person would consider issues such as these:

- *Applicability:* can the mechanism be used to express the required variation? Applicability is partly about the logical possibility of using a given mechanism: for example, it is not possible to use conditional compilation to implement a variation point that has a requirement to be bound at run time. Applicability also relates to the role of the mechanism in the ecosystem in which the software is being developed. For instance, although it is possible to use dynamic code generation to implement variability within a C program [7, 24], this is not common practice in C software.
- *Clarity:* how well does the mechanism capture the intent of the variation point? Beyond applicability, clarity of intent is about communicating the requirements of the variation point to human readers of the code. Clear expression of intent is related to the amount of code that the use of the variability mechanism will require (*conciseness*) and the distribution of that code (*modularity*, Section 1.2.2) throughout the implementation. It is also related to programming language idioms and design patterns: constructs that have emerged from the software community as means for encoding particular kinds of intent.
- *Consequences:* what are the effects of choosing the mechanism? One obvious effect is the human effort that will be needed to implement and maintain the variation point's implementation. This relates to concerns mentioned previously including conciseness, modularity, and the role of the mechanism in the programming ecosystem (e.g., tool support). A second effect is that the choice may entail, preclude, or otherwise involve the mechanisms that may be used for other variation points. A decision to use a particular remote procedure call [3] facility, for example, typically determines both an API to the facility and the formats of the messages that will be exchanged through the facility. A software designer may not want these decisions to be coupled, however. A third effect of a mechanism is its influence on properties of the software system overall. Although metrics such as source-code complexity and compiled program size can be important, variability mechanisms are often judged in terms of their effects on the run-time performance of software.

Software implementers often evaluate these issues in an implicit manner. Rather than select a variability mechanism based on an explicit list of requirements, it is common for a designer or programmer to choose a mechanism based on previously decided architectural qualities of the software being constructed, well-known programming idioms, well-known design patterns, and personal experience. *A result of this casual assessment is that the applicability, clarity, and consequences of well-known variability mechanisms are often seen as preset and fixed.* In some cases, this view may be warranted due to the limitations of the software-development tools that support the mechanisms. If all the available implementations of a particular variability mechanism work alike, then a decision to use the mechanism may entail certain consequences such as coupled decisions and run-time performance costs. In other cases, the view of fixed consequences is simply myopic. For example, many well-known design patterns are often seen as being applicable to object-oriented software only. When these patterns are viewed more abstractly—as definitions of entities, their roles, and their protocols—one can use them to guide the implementation of object-oriented and non-object-oriented software systems.

1.4 Novel Realizations of Variability Mechanisms

This dissertation seeks to demonstrate the benefits of thinking about well-known variability mechanisms in new ways. To that end, it focuses on three variability mechanisms that embody significant amounts of implementation detail: remote procedure calls, design patterns, and scheduling. These are representative mechanisms that embody some degree of abstraction over concrete implementation constructs.

Although these mechanisms are intended to provide architectural abstraction, software designers and programmers have a tendency to associate each of these mechanisms with particular implementation techniques. The result is that the applicability, clarity of intent, and consequences of a mechanism's stereotypical implementation becomes conflated with the applicability, clarity, and consequences of the mechanism itself. This dissertation shows the benefits that can be obtained by avoiding such confusion—by distinguishing between a variability mechanism and its stereotypical realization. A novel realization can make a mechanism applicable where it was not previously so. It can improve clarity of intent by promoting the concise and modular implementation of system requirements. Finally, novel realizations can mitigate or avoid the undesirable consequences that are associated with stereotypical implementations—in particular, run-time overheads.

The main part of this dissertation details three novel realizations of well-known variability mechanisms. Each is based on a well-known mechanism for modular software composition, and each modifies the composition mechanism while maintaining its modularity. Each demonstrates the ability of a novel realization to improve the run-time performance of a software system.

1.4.1 Flexible and Optimizing IDL Compilation

The first novel realization of a variability mechanism is *Flick*, a flexible and optimizing compiler for implementing remote procedure call (RPC) and remote method invocation (RMI) interfaces.

Flick is an example of an *interface definition language* (IDL) compiler. An IDL compiler reads an interface definition, written in an IDL, and outputs an implementation of that interface in a programming language such as C, C++, or Java. The implementation allows two or more agents—typically running in separate operating system processes, and perhaps running on different computers—to communicate through what appear to be ordinary procedure calls or object method invocations. The implementation of the interface hides the fact that the client of the interface (the caller) and the server of the interface (the callee) may be located in different processes. In other words, behind the facade of intra-process ("local") procedure call or method invocations, the code produced by an IDL compiler implements inter-process ("remote") communication. The implementations of the interface in the client and the server typically communicate by exchanging messages through a network protocol, e.g., a protocol based on TCP/IP.

As variability mechanisms, RPC and RMI provide several advantages.

Most obviously, RPC and RMI make it possible for the client and server of an interface to be connected at run time. This is in contrast to ordinary procedure-call interfaces within a program, which are typically bound and fixed at compile time or link time. Some implementations of RPC and RMI, such as those based in CORBA [22], even allow clients and servers to be connected, disconnected, and reconnected at run time. This allows for a high degree of flexibility in terms of binding features whose connections are realized via RPC or RMI.

Another potential advantage of RPC or RMI as a variability mechanism is that it may have relatively little impact on the implementation of other parts of the software system. Even when the clients and providers of a service are expected to reside within a single process, it is common software-engineering practice to separate the implementations of the clients and servers, and to clearly define an interface between them. Given this, implementing the interface via RPC or RMI may be relatively straightforward—but not necessarily trouble-free!

The disadvantages of the stereotypical realizations of RPC and RMI are threefold.

First, a typical IDL compiler provides little or no control over the interface to the code that it generates from an IDL specification. To compile an interface specification, an IDL compiler must map constructs in an IDL file (e.g., operations) onto constructs in another language (e.g., functions) that represent the original IDL constructs. The mapping from IDL constructs to target language constructs is the *presentation* of the interface in the target language. The presentation determines such things as the names of generated functions and the types of those functions' arguments and return values. In addition to syntactic issues, the presentation also encompasses semantic issues such as allocation protocols for memory. Interface definition languages such as CORBA IDL [22], the ONC RPC language [28], and the MIG language [23] are associated with standard mappings into languages such as C and C++. Those mappings are designed to be general and straightforward. Unfortunately, an unmalleable standard mapping can be a barrier to high-performance applications that rely on optimized communication channels. A "one size fits all" IDL compiler prevents programmers from using application-specific knowledge to optimize a system's performance.

Second, typical IDL compilers couple decisions that a designer or implementer may want to separate. The most significant of these deal with the communication infrastructure that the compiler-generated code will use. Some IDL compilers, such as MIG, support only a single underlying messaging system or a small and fixed set of such systems. A design-time decision to use such a compiler, therefore, becomes a design-time decision about the networking basis that the final software system will use. This may limit the applicability of such a compiler. Other IDL compilers, such as `rpcgen` and many implementations of CORBA, are supported by libraries that select a desired messaging subsystem at run time. Although this provides flexibility, it too represents a coupled choice: an implementation-time decision to use a traditional ONC RPC or CORBA compiler entails an implementation-time decision to *postpone* the choice of the transport system until run time. This lack of information at compile-time may impede the ability of an IDL compiler to produce optimized code.

Third, stereotypical IDL compilers make little effort to generate fast code. Many IDL compilers still assume that the transport medium is inherently slow, and therefore, that the generation of optimized code will not yield significant speed increases. Modern network architectures, however, have moved the performance bottlenecks for distributed applications out of the operating system layers and into the applications themselves [4, 14, 15]. The run-time overhead that is associated with stereotypical IDL compilers limits the perceived applicability of RPC and RMI as variability mechanisms in general.

Flick, our IDL compiler, addresses the disadvantages described above. Flick adapts concepts from traditional programming language compilers to bring both flexibility and optimization to the domain of IDL compilation.

To address the issues of inflexible language mappings and unnecessarily coupled decisions, Flick is designed as a set of components that may be specialized for particular IDLs, target-language mappings, data encodings, and transport mechanisms. It has front ends that parse the CORBA, ONC RPC, and MIG IDLs. Flick compiles an interface specification in any of these languages through a series of intermediate representations to produce CORBA-, `rpcgen`-, or MIG-style C code communicating via TCP, UDP, Mach [1] messages, or Fluke [10] kernel IPC. Flick's compilation stages are implemented as individual components, and it is easy for a system designer to select components at IDL compilation time in order to create the RPC or RMI implementation that he or she needs. The organization of Flick makes it straightforward to implement new component front ends, "presentation generators," and back ends.

To address the issue of run-time performance, Flick implements techniques such as code inlining, discriminator hashing, and careful memory management to maximize the speed at which data can be encoded and decoded for communication. Flick's optimization techniques are similar to those provided by modern optimizing compilers, but its domain-specific knowledge allows Flick to implement important optimizations that a general-purpose language compiler cannot.

Chapter 2 details the design, implementation, and evaluation of Flick. By addressing both flexibility and optimization concerns, Flick widens the potential applicability of RPC and RMI as variability mechanisms for software design.

1.4.2 Static and Dynamic Structure in Design Patterns

The second area of novel variation mechanism realizations is based on *design patterns*. A design pattern is a reusable solution to a recurring problem in software design [13]. A pattern is meant to describe a general solution technique that can be applied to a particular kind of situation in software design. In contrast to describing the implementation-level details of a particular solution within a particular software system, a pattern describes the general structure of a problem and a solution, so that the solution can be adapted and implemented as needed for solving many instances of a given design problem. The problem is described in terms of design issues; the solution is described in terms of participating entities, their roles, and how they cooperate to address the problem.

Many design patterns describe ways of organizing variability in software, and thus, the implementations of patterns are often used as variation mechanisms. As detailed in Chapter 3, however, the stereotypical view of design patterns limits their application in practice.

The conventional approach to realizing patterns [13] primarily uses classes and objects to implement participants and uses inheritance and object references to implement relationships between participants. The parts of patterns that are realized by classes and inheritance correspond to static information about the software—information that can be essential for understanding, checking, and optimizing a program. Unfortunately, class structures can disguise the underlying pattern relationships, both by being too specific (to a particular application of a pattern) and by being mixed with unrelated code. In contrast, the parts of patterns realized by run-time objects and references are more dynamic and flexible, but are therefore harder to understand and analyze.

More generally, software designers and programmers commonly associate design patterns with their stereotypical implementations in object-oriented languages, which involve classes and objects as described previously. This means that patterns may not applied in situations in which they might be helpful—i.e., in the design of non-object-oriented systems. The stereotypical implementation of patterns using run-time objects also has certain consequences, such as potential run-time performance overheads. These costs can potentially be mitigated or avoided when necessary, however, through novel realizations of design patterns.

Chapter 3 describes such a novel approach to realizing patterns, one based on separating the static parts of a pattern from the dynamic parts. The *static participants* and relationships in a pattern are realized by component instances and component interconnections that are set at compile- or link-time, while the *dynamic participants* continue to be realized by objects and object references. Expressing static pattern relationships as component interconnections provides more flexibility than the conventional approach while also promoting ease of understanding and analysis.

The basis of the approach is to permit system configuration and realization of design patterns at compile- and link-time (i.e., before software is deployed) rather than at initialization- and run-time (i.e., after it is deployed). Components are defined and connected in a language that is separate from the implementation language of a software system, thus allowing one to separate configuration concerns from the implementation of a system's parts. A system can be reconfigured at the level of components, possibly by a nonexpert, and can be analyzed to check design rules or optimize the overall system. This approach helps a programmer to identify design trade-offs and strike an appropriate balance between design-time and run-time flexibility. More generally, the approach described in Chapter 3 demonstrates the usefulness of a novel realization of variability mechanisms—design patterns—for implementing systems in which run-time performance is a concern.

1.4.3 Dynamic CPU Management for Real-Time Systems

The third novel realization of a variability mechanism is the *CPU Broker*, a facility for mediating between multiple real-time tasks and the facilities of a real-time operating system. The CPU Broker allows the tasks of a real-time system to be usefully composed late in the software life cycle, i.e., after the individual tasks have been implemented, and possibly after they have been delivered to a customer and deployed on a computer. In other words, the features (individual tasks) of the real-time system can be selected and usefully composed. The CPU Broker supports such late composition by connecting to its managed tasks in a noninvasive manner. During system execution, using feedback and other inputs, the broker adjusts the CPU allocations of its managed tasks to ensure that high application-level quality-of-service (QoS) is maintained.

The CPU Broker is designed to apply to multi-agent real-time systems that are built atop commercial, off-the-shelf (COTS) real-time operating systems and middleware. Even with modern systems as a basis, it can be a significant challenge for system developers to design and build real-time systems that meet their overall requirements for performance.

- Because the parts of a software system must often be designed to be reusable across many products, the code that implements real-time behavior for any particular system must be decoupled from the "application logic" of the system's parts. Decoupling makes it possible to collect the real-time specifications for all of the system's parts in one place, but leads to the problem of reintroducing that behavior into the software.
- Even if the implementation of real-time behavior is modularized, developers are challenged with specifying the desired behavior at all. It is common for the execution times of parts of a system to be data-dependent, mode-dependent, configuration-dependent, unpredictable, or unknown. In a multi-agent real-time system, the sets of communicating tasks and available processor resources may not be known until run time, or may change as the system is running.

Thus, the challenges of implementing real-time behavior in multi-agent systems include not only (1) decoupling and modularizing of the behavior, but also (2) describing a variety of policies in a high-level and tractable manner and (3) ensuring that the system continues to operate—perhaps at reduced capacity—in the face of events that occur at run time, both expected and unexpected.

The CPU Broker described in Chapter 4 addresses these challenges. It is designed to ensure that the CPU demands of "important" applications are satisfied insofar as possible, especially in the face of dynamic changes in resource requirements and availability, in the set of managed tasks, and in the relative importances of the tasks.

The CPU Broker is a CORBA-based server that mediates between the multiple real-time tasks and the facilities of a real-time operating system, such as TimeSys Linux [32]. The broker addresses design-time challenges by connecting to its managed tasks in a noninvasive fashion and by providing an expressive and open architecture for specifying CPU scheduling policies. The broker can manage resources for both CORBA and non-CORBA applications. At run time, the broker uses feedback and other inputs to monitor resource usage, adjust allocations, and deal with contention according to a configured policy or set of policies. The broker is configured at run time through a command-line tool or via invocations on the CORBA objects within the broker: policies are easily set up and changed dynamically. The experimental results presented in Chapter 4 show that the broker approach can effectively address both the design-time and run-time challenges of managing real-time behavior in COTS-based real-time systems.

The broker demonstrates the benefits that can be obtained through a novel implementation of scheduling, which is an essential part of a variation mechanism that divides a large system into separate tasks. By implementing a centralized and programmable "negotiation" facility atop a COTS real-time scheduler, the CPU Broker increases the effectiveness of scheduling for realizing multi-agent, real-time systems in which the agents, their CPU requirements, or their relative importances are determined late in the software life cycle.

1.5 References

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer 1986 USENIX Conf.* (June 1986), pp. 93–112.

[2] Anastasopoulos, M., and Gacek, C. Implementing product line variabilities. In *Proc. of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context (SSR)* (Toronto, ON, May 2001), pp. 109–117.

[3] Birrell, A. D., and Nelson, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39–59.

[4] Clark, D. D., and Tennenhouse, D. L. Architectural considerations for a new generation of protocols. In *Proc. of the SIGCOMM '90 Symp.* (1990), pp. 200–208.

[5] Clements, P., and Northrop, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[6] Czarnecki, K., and Eisenecker, U. W. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[7] Engler, D. R. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proc. of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, May 1996), pp. 160–170.

[8] Ernst, M. D., Badros, G. J., and Notkin, D. An empirical study of C preprocessor use. *IEEE Transactions on Software Engineering 28*, 12 (Dec. 2002), 1146–1170.

[9] Filman, R. E., Elrad, T., Clarke, S., and Akşit, M. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[10] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. Microkernels meet recursive virtual machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation* (Seattle, WA, Oct. 1996), USENIX Assoc., pp. 137–151.

[11] Fowler, M. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, Jan. 2004.

[12] Fritsch, C., Lehn, A., and Strohm, T. Evaluating variability implementation mechanisms. In *Proc. of the PLEES '02 International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing* (Oct. 2002), K. Schmid and B. Geppert, Eds., pp. 59–64. Fraunhofer IESE-Report No. 056.02/E.

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

[14] Gokhale, A., and Schmidt, D. C. Measuring the performance of communication middleware on high-speed networks. *Computer Communication Review 26*, 4 (Oct. 1996).

[15] Gokhale, A., and Schmidt, D. C. Optimizing the performance of the CORBA Internet Inter-ORB Protocol over ATM. Tech. Rep. WUCS–97–09, Washington University Department of Computer Science, St. Louis, MO, 1997.

[16] Jacobson, I., Griss, M., and Jonsson, P. *Software Reuse: Architecture, Process, and Organization for Business Success*. Addison-Wesley, 1997.

[17] Johnson, R. E., and Foote, B. Designing reusable classes. *Journal of Object-Oriented Programming 1*, 2 (June/July 1988), 22–35.

[18] Lohmann, D., Hofer, W., Schröder-Preikschat, W., Streicher, J., and Spinczyk, O. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proc. of the 2009 USENIX Annual Technical Conference* (San Diego, CA, June 2009), pp. 215–228.

[19] Mecklenburg, R. *Managing Projects with GNU Make*. O'Reilly Media, Inc., 2004.

[20] Muller, G., Marlet, R., Volanschi, E.-N., Consel, C., Pu, C., and Goel, A. Fast, optimized Sun RPC using automatic program specialization. In *Proc. of the 18th International Conference on Distributed Computing Systems (ICDCS)* (Amsterdam, The Netherlands, May 1998), pp. 240–249.

[21] Muller, G., Volanschi, E.-N., and Marlet, R. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *Proc. of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)* (Amsterdam, The Netherlands, June 1997), pp. 116–126.

[22] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[23] Open Software Foundation and Carnegie Mellon University. *Mach 3 Server Writer's Guide*. Cambridge, MA, Jan. 1992.

[24] Poletto, M., Hsieh, W. C., Engler, D. R., and Kaashoek, M. F. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems 21*, 2 (Mar. 1999), 324–369.

[25] Scott, M. L. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, 2000.

[26] Singh, N., Gibbs, C., and Coady, Y. C-CLR: A tool for navigating highly configurable system software. In *Proc. of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)* (Vancouver, BC, Mar. 2007).

[27] Spencer, H., and Collyer, G. #ifdef considered harmful, or portability experience with C News. In *Proc. of the USENIX Summer 1992 Technical Conference* (San Antonio, TX, June 1992), pp. 185–197.

[28] Srinivasan, R. RPC: Remote procedure call protocol specification version 2. Tech. Rep. RFC 1831, Sun Microsystems, Inc., Aug. 1995.

[29] Svahnberg, M., van Gurp, J., and Bosch, J. A taxonomy of variability realization techniques. *Software: Practice and Experience 35*, 8 (July 2005), 705–754.

[30] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, second ed. Addison-Wesley, 2002.

[31] Taylor, R. N., Medvidović, N., and Dashofy, E. M. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, Inc., 2010.

[32] TimeSys Corporation. TimeSys Linux GPL: Performance advantages for embedded systems. White paper, version 1.1., 2003.

[33] Vaughan, G. V., Elliston, B., Tromey, T., and Taylor, I. L. *GNU Autoconf, Automake, and Libtool*. New Riders Publishing, 2000.

PART I

FLICK: A FLEXIBLE, OPTIMIZING IDL COMPILER

Reprinted from *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 44–56, June 1997: "Flick: A Flexible, Optimizing IDL Compiler" by Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. © 1997 Association for Computing Machinery, Inc. Reprinted by permission. `http://doi.acm.org/10.1145/258915.258921`

CHAPTER 2

FLICK: A FLEXIBLE, OPTIMIZING IDL COMPILER

An interface definition language (IDL) is a nontraditional language for describing interfaces between software components. IDL compilers generate "stubs" that provide separate communicating processes with the abstraction of local object invocation or procedure call. High-quality stub generation is essential for applications to benefit from component-based designs, whether the components reside on a single computer or on multiple networked hosts. Typical IDL compilers, however, do little code optimization, incorrectly assuming that interprocess communication is always the primary bottleneck. More generally, typical IDL compilers are "rigid" and limited to supporting only a single IDL, a fixed mapping onto a target language, and a narrow range of data encodings and transport mechanisms.

Flick, our new IDL compiler, is based on the insight that IDLs are true languages amenable to modern compilation techniques. Flick exploits concepts from traditional programming language compilers to bring both flexibility and optimization to the domain of IDL compilation. Through the use of carefully chosen intermediate representations, Flick supports multiple IDLs, diverse data encodings, multiple transport mechanisms, and applies numerous optimizations to all of the code it generates. Our experiments show that Flick-generated stubs marshal data between 2 and 17 times faster than stubs produced by traditional IDL compilers, and on today's generic operating systems, increase end-to-end throughput by factors between 1.2 and 3.7.

2.1 Introduction

An *interface definition language* (IDL) is a special-purpose language for describing the interfaces of a software component. An IDL specification declares one or more interfaces; each interface declares a set of operations that may be invoked on objects implementing the interface. The input and output behavior of each operation is given by the IDL specification.

For example, the following CORBA [18] IDL program declares a simple interface to an electronic mail service:

```
interface Mail {
    void send(in string msg);
};
```

A largely equivalent mail system interface would be defined in the ONC RPC[1] [23] IDL by this program:

```
program Mail {
    version MailVers {
        void send(string) = 1;
    } = 1;
} = 0x20000001;
```

As shown by these examples, an IDL program declares a set of functions or methods but does not describe the computations that those functions and methods perform. IDLs are typically independent of the programming language in which the components are themselves implemented, further decoupling interface from implementation.

An IDL compiler accepts an IDL interface specification and outputs an implementation of that specification. Typically, the implementation is a set of data type declarations and "stubs" written in a conventional programming language such as C, C++, or Java. The stubs encapsulate the communication that must occur between the entity that invokes an operation (i.e., the *client*) and the entity that implements the operation (i.e., the *server*). The stubs that are output by the IDL compiler hide the details of communication and allow the client and server to interact through a procedural interface. Traditionally, stubs have implemented *remote procedure calls* (RPC) [3] or *remote method invocations* (RMI): the client and server are located in separate processes, and the stubs in each process communicate by exchanging messages through a transport medium such as TCP/IP. More recently, IDLs have become popular for defining high-level interfaces between program modules within a single process.

[1]ONC RPC was previously known as Sun RPC, and Sun's `rpcgen` is the standard compiler for the ONC RPC IDL. The numbers in the example ONC RPC IDL program are chosen by the programmer to identify components of the interface.

IDLs and IDL compilers arose for reasons familiar to any programming language veteran: descriptive clarity, programmer productivity, assurance of consistency, and ease of maintenance. Performance of IDL-generated code, however, has traditionally not been a priority. Until recently, poor or mediocre performance of IDL-generated code was acceptable in most applications: because interprocess communication was generally both expensive and rare, it was not useful for an IDL compiler to produce fast code. For performance critical applications, implementors resorted to hand-coded stubs—tolerating the accompanying greater software engineering costs. Some IDL compilers such as MIG [20] struck a middle ground by providing a language with a restricted set of structured data types, blended with programmer control over implementation details. This compromise could be likened to that provided by a traditional compiler that permits embedded assembly language. Although embedded "hints" can lead to performance gains, reliance on hints moves the burden of optimization from the compiler to the programmer, and has the additional effect of making the language non-portable or useful only within restricted domains.

Today, in almost every respect, IDL compilers lag behind traditional language compilers in terms of flexibility and optimization. IDL compilers such as Sun's `rpcgen` [25] are generally written "from scratch" and are implemented without incorporating modern compiler technologies such as multiple, flexible intermediate representations. The result is that today's IDL compilers are "rigid": they accept only a single IDL, they implement only a single, fixed mapping from an IDL specification to a target language, and they generate code for only one or two encoding and transport mechanisms. Today's IDL compilers still assume that the transport medium is inherently slow, and therefore, that optimization of the stubs will not yield significant speed increases. Modern network architectures, however, have moved the performance bottlenecks for distributed applications out of the operating system layers and into the applications themselves [5, 12, 13].

In this paper we show that in order to solve the problems inherent to existing IDL compilers, IDL compilation must evolve from an ad hoc process to a principled process incorporating techniques that are already well-established in the traditional programming language community. Although IDL compilation is a specialized domain, IDL compilers can be greatly improved through the application of concepts and technologies developed for the compilation of general programming languages. *Flick*, our Flexible IDL Compiler Kit, exploits this idea. Flick is designed as a "toolkit" of reusable components that may be specialized for particular IDLs, target language mappings, data encodings, and transport mechanisms. Flick currently has front ends that parse the CORBA [18], ONC RPC [23], and MIG [20] IDLs. Flick compiles an interface specification in any of these languages through a series of intermediate representations to produce CORBA-, `rpcgen`-, or MIG-style C stubs communicating via TCP, UDP, Mach [1] messages, or Fluke [10] kernel IPC. Flick's compilation stages are implemented as individual components and it is easy for a system designer to mix and match components at IDL compilation time in order to create the high-performance communication stubs that he or she needs. Further, the organization of Flick makes it easy to implement new component front ends, "presentation generators," and back ends.

Flick's design as a traditional language compiler promotes not only flexibility but also optimization. Flick implements techniques such as code inlining, discriminator hashing, and careful memory management to maximize the speed at which data can be encoded and decoded (*marshaled* and *unmarshaled*) for communication. Flick's optimization techniques are similar to those provided by modern optimizing compilers, but its domain-specific knowledge allows Flick to implement important optimizations that a general-purpose language compiler cannot. Most of Flick's techniques are implemented by an abstract C++ base class for code generators, and therefore, all back ends inherit the optimizations provided by the large code base. The results presented in Section 2.4 show that Flick-generated communication stubs are up to 3.7 times faster than those generated by other IDL compilers.

2.2 Flick

The Flick IDL compiler is divided into three phases as illustrated in Figure 2.1. These phases are analogous to those in a traditional language compiler and correspond to separable aspects of IDL compilation. Each phase is primarily implemented by a large, shared library of C and C++ code that provides abstractions for such things as IDL source constructs, target language data types, and "on the wire" message data types. Each of Flick's libraries implements a generic set of methods to manipulate these abstractions. The libraries are the bases for specializations that override the generic methods as necessary in order to implement behaviors peculiar or specific to a single IDL, language mapping, message format, or transport facility.

The first phase of the compiler is the *front end*. The front end reads an IDL source file

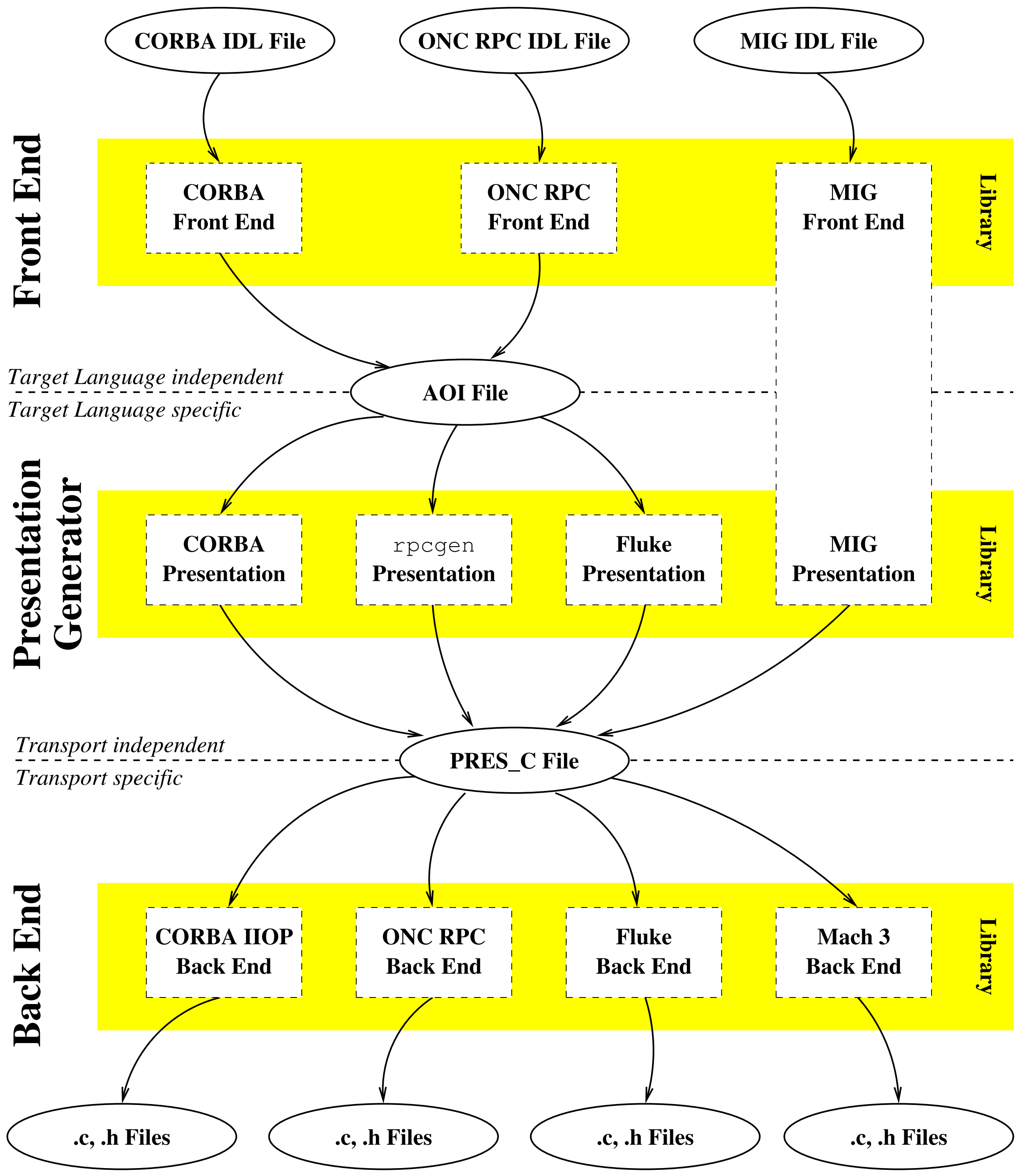


Figure 2.1. Overview of the Flick IDL compiler. Flick is divided into three compilation phases, and each phase is implemented by a large library of code. Specialized components are derived from the Flick libraries in order to parse different IDLs, implement different target language mappings, and produce code for a variety of message formats and transport systems.

and produces an abstract representation of the interface defined by the IDL input. This representation, called an Abstract Object Interface (AOI), describes the high-level "network contract" between a client and a server: the operations that can be invoked and the data that must be communicated for each invocation.

Flick's second compilation phase, the *presentation generator*, reads the network contract produced by the front end and outputs a separate and lower-level "programmer's contract." The programmer's contract defines the interface between the programmer's client or server code and the stubs, e.g., how parameters are passed between them.

For example, consider the CORBA IDL input shown in Section 2.1 that defines a network contract between the client and server of a `Mail` interface. Given that input, a CORBA IDL compiler for C will always produce the following prototype describing the programmer's contract:[2]

```
void Mail_send(Mail obj, char *msg);
```

This programmer's contract declares the C functions and data types that will connect the client or server code to the stub: we say that this contract is a *presentation* of the interface in the C language.

The presentation shown above conforms to the CORBA specification for mapping IDL constructs onto the C programming language. However, it is not the only possible presentation of the `Mail` interface. For instance, if we depart from the CORBA mapping rules, the `Mail_send` function could be defined to take a separate message length argument:

```
void Mail_send(Mail obj, char *msg, int len);
```

This presentation of the `Mail` interface could enable optimizations because `Mail_send` would no longer need to count the number of characters in the message [8, 9]. This change to the presentation would *not* affect the network contract between client and server; the messages exchanged between client and server would be unchanged. The addition of a separate `len` parameter changes *only* the calling conventions for the `Mail_send` function. Flick's ability to handle different presentation styles can be important for optimization as just described, but it is also essential for supporting multiple IDLs in a reasonable way.

[2] For clarity, we have omitted the declaration of the `Mail` object type and the `CORBA_Environment` parameter to the `Mail_send` function.

To summarize, a *presentation* describes everything that client or server code must understand in order to use the function and data type declarations output by an IDL compiler: this includes the names of the functions, the types of their arguments, the conventions for allocating memory, and so on. Because there can be many different presentations of a single interface, Flick provides multiple, different *presentation generators*, each implementing a particular style of presentation for a particular target programming language. When C is the target language, the presentation is described in an intermediate format called PRES_C. Because the presentation of an interface may differ between client and server, a presentation generator creates separate PRES_C files for the client- and server-side presentations of an interface.

The third and final phase of the compiler is the *back end*. The back end reads a presentation description (PRES_C) and produces the source code for the C functions that will implement client/server communication. The generated functions are specific to a particular message format, message data encoding scheme, and transport facility.

Table 2.1 compares the number of substantive C and C++ source code lines in each of Flick's libraries with the number of lines particular to each of Flick's specialized components. The number of lines specific to each presentation generator and back end is extremely small when compared to the size of the library from which it is derived. Front

Table 2.1. Code reuse within the Flick IDL compiler. Percentages show the fraction of the code that is unique to a component when it is linked with the code for its base library. The CORBA presentation library is derived from the generic presentation library; the CORBA and Fluke presentation generators are derived from the CORBA presentation library.

Phase	Component	Lines	
Front End	Base Library	1797	
	CORBA IDL	1661	48.0%
	ONC RPC IDL	1494	45.4%
Pres. Gen.	Base Library	6509	
	CORBA Library	770	10.6%
	CORBA Pres.	3	0.0%
	Fluke Pres.	301	4.0%
	ONC RPC `rpcgen` Pres.	281	4.1%
Back End	Base Library	8179	
	CORBA IIOP	353	4.1%
	ONC RPC XDR	410	4.8%
	Mach 3 IPC	664	7.5%
	Fluke IPC	514	5.9%

ends have significantly greater amounts of specialized code due to the need to scan and parse different IDL source languages.

2.2.1 Front Ends

As just described, the purpose of a Flick front end is to translate an interface description (source IDL program) to an intermediate representation. Each of Flick's front ends is specific to a particular IDL. However, each is completely *independent* of the later stages of IDL compilation: the presentation of the interface that will be constructed, the target programming language that will implement the presentation, the message format and data encodings that will be chosen, and the transport mechanism that will be used. In sum, the output of a Flick front end is a high-level "network contract" suitable for input to any presentation generator and any back end.

Flick's MIG front end, however, is a special case. A MIG interface definition contains constructs that are applicable only to the C language and to the Mach message and IPC systems [20]. Therefore, as illustrated in Figure 2.1, Flick's MIG front end is conjoined with a special MIG presentation generator that understands these idioms. Flick's MIG components translate MIG interface descriptions directly into PRES_C representations; this is different than Flick's CORBA and ONC RPC front ends, which produce AOI. This difference reveals a strength: Flick's multiple intermediate representations provide the flexibility that is necessary for supporting a diverse set of IDLs.

2.2.1.1 AOI: The Abstract Object Interface

AOI is Flick's intermediate representation language for describing interfaces: the data types, operations, attributes, and exceptions defined by an IDL specification. AOI is applicable to many IDLs and represents interfaces at a very high level. It describes constructs independently of their implementation: for instance, AOI has separate notions of object methods, attributes, and exceptions, although all of these things are generally implemented as kinds of messages. AOI supports the features of typical existing IDLs such as the CORBA and ONC RPC IDLs, and Flick's front ends produce similar AOI representations for equivalent constructs across different IDLs. This "distillation process" is what makes it possible for Flick to provide a large and general library for the next stage of compilation, presentation generation.

2.2.2 Presentation Generators

Presentation generation is the task of deciding how an interface description will be mapped onto constructs of a target programming language. Each of Flick's presentation generators implements a particular mapping of AOI constructs (e.g., operations) onto target language constructs (e.g., functions). Therefore, each presentation generator is specific to a particular set of mapping rules and a particular target language (e.g., the CORBA C language mapping).

A presentation generator determines the appearance and behavior (the "programmer's contract") of the stubs and data types that present an interface—but *only* the appearance and behavior that is exposed to client or server code. The unexposed *implementation* of these stubs is determined later by a Flick back end. Therefore, the function definitions produced by a presentation generator may be implemented on top of any available transport facility, and each presentation generator is independent of any message encoding or transport. Moreover, each of Flick's presentation generators (except for the MIG generator as described previously) is independent of any particular IDL. A single presentation generator can process AOI files that were derived from several different IDLs.[3]

Flick currently has two presentation generators that read AOI files: one that implements the C mapping specified by CORBA [18] and a second that implements the C mapping defined by Sun Microsystems' `rpcgen` program [25]. Each of these presentation generators outputs its presentations in an intermediate representation called PRES_C (Presentation in C). PRES_C is a fairly complex description format containing three separate sublanguages as illustrated in Figure 2.2 (and described separately below): a *MINT* representation of the messages that will be exchanged between client and server, a *CAST* representation of the output C language declarations, and a set of *PRES* descriptions that connect pieces of the CAST definitions with corresponding pieces of the MINT structures. Of the three intermediate representations within a PRES_C file, only CAST is specific to the C language; MINT and PRES are applicable to any programming language. We plan to create intermediate representation languages for C++ and Java presentations, for example,

[3]Naturally, the ability to process AOI files generated from different IDLs is somewhat restricted due to the limitations of particular presentations. For example, the presentation generator that implements the `rpcgen` presentation style cannot accept AOI files that use CORBA-style exceptions because there is no concept of exceptions in standard `rpcgen` presentations. Similarly, the CORBA presentation generator cannot handle self-referential type definitions that may occur in an AOI file produced from an ONC RPC IDL input because CORBA does not support self-referential types.

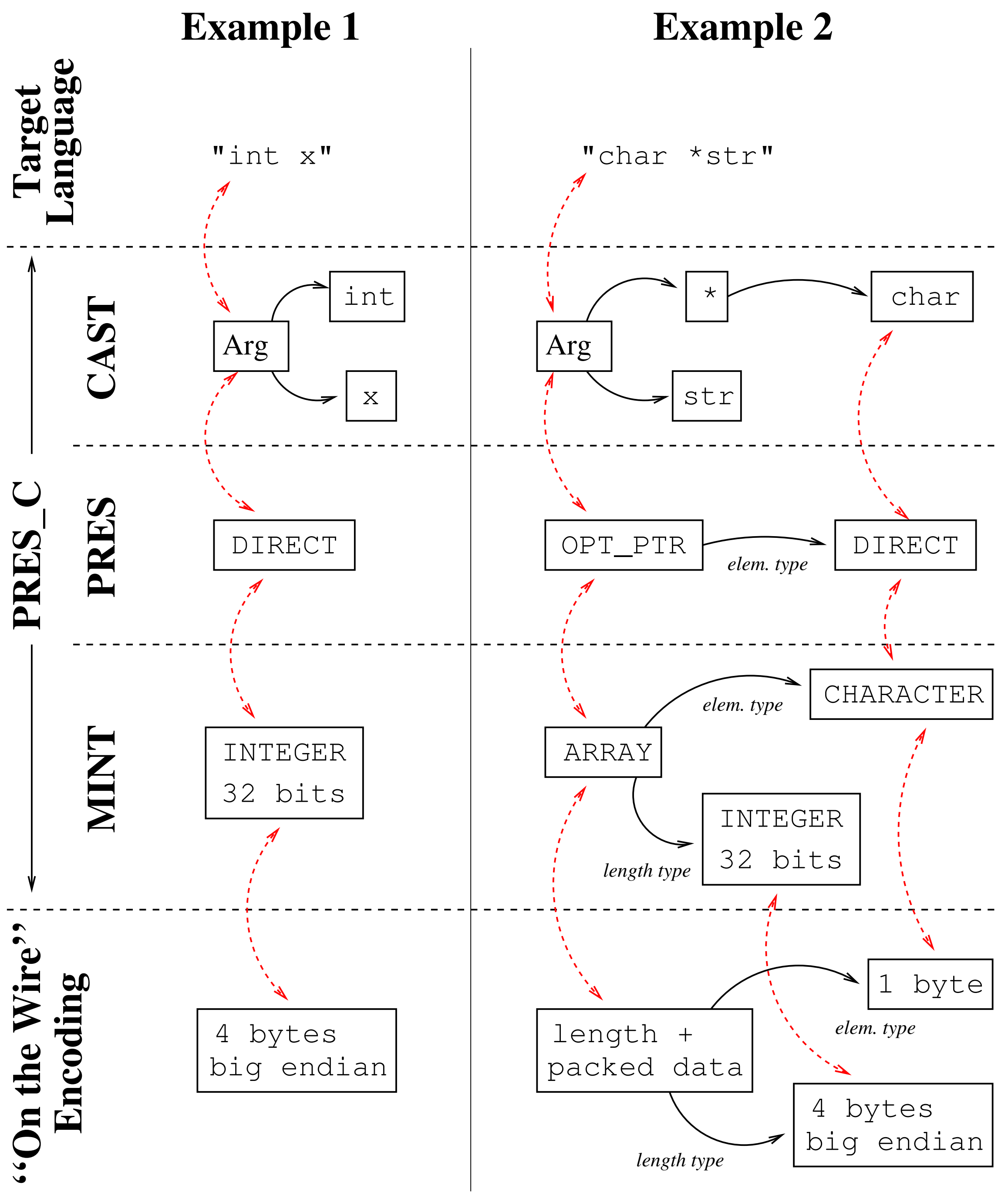


Figure 2.2. Two examples of PRES_C. PRES_C is the intermediate representation that connects C target language data with “on the wire” data encodings. The first example links a C language `int` with a 4-byte, big-endian encoding. The second example associates a C string (`char *`) with a counted array of packed characters.

by replacing CAST with intermediate representation languages for C++ and Java source code.

2.2.2.1 MINT: The Message Interface

The first step of presentation generation is to create an abstract description of all messages, both requests and replies, that may be exchanged between client and server as part of an interface. These messages are represented in a type description language called MINT. A MINT representation of a data type is a directed graph (potentially cyclic) with each node representing an atomic type (e.g., an integer), an aggregate type (e.g., a fixed- or variable-length array, structure, or discriminated union), or a typed literal constant.

MINT types do not represent types in the target programming language, nor do they represent types that may be encoded within messages. Rather, MINT types represent high-level message formats, describing all aspects of an “on the wire” message *except* for low-level encoding details. MINT types serve as glue between transport encoding types and target language types as illustrated in Figure 2.2. The first example in Figure 2.2 utilizes a MINT integer type that is defined to represent signed values within a 32-bit range. The MINT integer type does not specify any particular encoding of these values, however. Target language issues are specified by the representation levels above MINT in the figure; “on the wire” data encodings are specified by the representation level below MINT. The second example in Figure 2.2 illustrates a MINT array type containing both a length and a vector of characters. Again, MINT specifies the ranges of the values within the type but does not specify any encoding or target language details.

2.2.2.2 CAST: The C Abstract Syntax Tree

The second portion of a PRES_C file is a description of the C language data types and stubs that present the interface. These constructs are described in a language called CAST, which is a straightforward, syntax-derived representation for C language declarations and statements. By keeping an explicit representation of target language constructs, Flick can make associations between CAST nodes and MINT nodes described previously. Explicit representation of target language constructs is critical to flexibility; this is the mechanism that allows different presentation generators and back ends to make fine-grain specializations to the base compiler libraries. Similarly, explicit representation is critical to optimization because Flick’s back ends must have complete associations between target language data and "on the wire" data in order to produce efficient marshaling and unmarshaling code.

Although Flick's explicit representation for C language constructs is ordinary in comparison to the intermediate representations used by traditional language compilers, it is unique in comparison to traditional IDL compilers because most IDL compilers including `rpcgen` and ILU [15] maintain no explicit representations of the code that they produce.

2.2.2.3 PRES: The Message Presentation

PRES, the third and final component of PRES_C, defines the mapping between the message formats defined in MINT and the target language-specific, application-level formats defined in CAST. Like MINT and CAST, PRES is a graph-based description language. A node in a PRES tree describes a relationship between a MINT node and a CAST node: the data described by the MINT and CAST nodes are "connected" and marshaling and unmarshaling of data will take place as determined by the connecting PRES node. In language terms, a PRES node defines a *type conversion* between a MINT type and a target language type.

Different PRES node types describe different styles of data presentation as illustrated in Figure 2.2. In the first example, a MINT integer is associated with a C language integer through a direct mapping: no special data transformation is specified. In the second example, a MINT variable-length array is associated with a C pointer. The PRES node is an OPT_PTR node and specifies that a particular kind of transformation must occur for both data marshaling and unmarshaling. Consider the unmarshaling case. The OPT_PTR node defines that when the MINT array size is nonzero, the array elements will be unmarshaled and the C pointer will be set to point at the decoded array elements—in this example, characters. If the MINT array size is zero, the C pointer will be set to null. Reverse transformations occur when the C pointer data must be marshaled into a message.

Other PRES node types define similar kinds of presentation styles, and the set of PRES node types is designed to cover all of the transformations required by existing presentation schemes. PRES is not specific to any one programming language, although certain node types depend on certain language features. For instance, OPT_PTR nodes only make sense for target languages that have pointers.

2.2.2.4 PRES_C: The C Presentation

PRES_C combines the intermediate representations described above to create a complete description language for C language interface presentations. A PRES_C file contains the array of stub declarations that will present the interface (to the client or server, not both). Each stub is associated with its declaration in CAST, the MINT description of the messages it receives, the MINT description of the messages it sends, and two PRES trees that associate pieces of the two MINT trees with the function's CAST declaration.

In total, a PRES_C file is a complete description of the presentation of an interface—it describes everything that a client or server must know in order to invoke or implement the operations provided by the interface. The only aspect of object invocation not described by PRES_C is the transport protocol (message format, data encoding, and communication mechanism) that will be used to transfer data between the client and the server. This final aspect of IDL compilation is the domain of Flick's back ends.

2.2.3 Back Ends

A Flick *back end* inputs a description of a presentation and outputs code to implement that presentation in a particular programming language. For presentations in C, the input to the back end is a PRES_C file and the output is a "`.c`" file and a corresponding "`.h`" file. The output C code implements the interface presentation for either the client or the server. Because the output of a presentation generator completely describes the appearance and exposed behavior of the stubs that implement an interface, Flick's back ends are entirely independent of the IDL and presentation rules that were employed to create a presentation.

Each back end is, however, specific to a single programming language, a particular message encoding format, and a particular transport protocol. All of the currently implemented back ends are specific to C, but Flick's "kit" architecture will support back ends specific to other languages such as C++ or Java in the future. Each of Flick's C back ends supports a different communication subsystem: the first implements the CORBA IIOP (Internet Inter-ORB Protocol) [18] on top of TCP; the second sends ONC RPC messages [23, 24] over TCP or UDP; the third supports MIG-style typed messages sent between Mach 3 ports; and the fourth implements a special message format for the fast Fluke kernel IPC facility [10]. Although these four communication subsystems are all very different, Flick's back ends share a large library of code to optimize the marshaling and unmarshaling of data. This library operates on the MINT representations of the messages.

Whereas a presentation generator creates associations between MINT types and target language types (through PRES), a back end creates associations between MINT types and "on the wire" encoding types. The mapping from message data to target language is therefore a chain: from encoded type to MINT node, from MINT node to PRES node, and from PRES node to CAST. Flick's library for C back ends operates on these chains and performs optimizations that are common to all transport and encoding systems.

2.3 Optimization

Flick's back ends apply numerous domain-specific optimization techniques to address the performance problems that typically hinder IDL-based communication. Flick's optimizations are complementary to those that are generally implemented by traditional language compilers. While many of Flick's techniques have counterparts in traditional compilers, Flick is unique in that it has knowledge of its specialized task domain and has access to many different levels of information through its multiple intermediate representations. This allows Flick to implement optimizations that a general language compiler cannot. Conversely, Flick produces code with the expectation that general optimizations (e.g., register allocation, constant folding, and strength reduction) will be performed by the target language compiler. In summary, Flick implements optimizations that are driven by its task domain and delegates general-purpose code optimization to the target language compiler.

2.3.1 Efficient Memory Management

2.3.1.1 Marshal Buffer Management

Before a stub can marshal a datum into its message buffer, the stub must ensure that the buffer has at least enough free space to contain the encoded representation of the datum. The stubs produced by typical IDL compilers check the amount of free buffer space before *every* atomic datum is marshaled, and if necessary, expand the message buffer. These repeated tests are wasteful, especially if the marshal buffer space must be continually expanded. The stubs produced by Flick avoid this waste.

Flick analyzes the overall storage requirements of every message that will be exchanged between client and server. This is accomplished by traversing the MINT representation of each message. The storage requirements and alignment constraints for atomic types are given by the "on the wire" data types that are associated with the various MINT nodes. The storage requirements for aggregate types are determined by working backward from nodes with known requirements. Ultimately, Flick classifies every type into one of three storage size classes: fixed, variable and bounded, or variable and unbounded.

From this information, Flick produces optimized code to manage the marshal buffer within each stub. Before marshaling a fixed-size portion of a message, a Flick-generated stub will ensure that there is enough free buffer space to hold *all* of the component data within the fixed-size message segment. The code that actually marshals the data is then free to assume that sufficient buffer space is available. In cases in which an entire message has a fixed size, Flick generates a stub that checks the size of the marshal buffer exactly once.[4] A different message segment may be variable in size but bounded by a limit known at stub generation time or by a limit known at stub execution time. In this case, if the range of the segment size is less than a predetermined threshold value (e.g., 8KB), Flick produces code similar to that for fixed-size message fragments: the generated stub will ensure that there is enough space to contain the maximum size of the message segment. If the range is above the threshold, however, or if the message fragment has no upper bound at all, then Flick "descends" into the message segment and considers the segment's subcomponents. Flick then analyzes the subcomponents and produces stub code to manage the largest possible fixed-size and threshold-bounded message segments as described above. Overall, our experiments with Flick-generated stubs have shown that this memory optimization technique reduces marshaling times by up to 12% for large messages containing complex structures.

2.3.1.2 Parameter Management

Another optimization that requires domain knowledge is the efficient management of memory space for the parameters of client and server stubs. Just as it is wasteful to allocate marshal buffer space in small pieces, it is similarly wasteful to allocate memory for unmarshaled data on an object-by-object or field-by-field basis. Therefore, Flick optimizes the allocation and deallocation of memory used to contain the unmarshaled data that will be presented to clients and servers. For example, Flick-generated stubs may use the runtime stack to allocate space for parameter data when this is allowed by the semantics of the interface presentation. In some situations, Flick-generated stubs use space within the

[4]Flick-generated stubs use dynamically allocated buffers and reuse those buffers between stub invocations. This is generally preferable to allocating a new buffer for each invocation. However, it means that stubs that encode fixed-size messages larger than the minimum buffer size must verify the buffer size once.

marshal buffer itself to hold unmarshaled data—this optimization is especially important when the encoded and target language data formats of an object are identical. Generally, these optimizations are valid only for `in` (input) parameters to the functions in a server that receive client requests. Further, the semantics of the presentation must forbid a server function from keeping a reference to a parameter's storage after the function has returned. Our experiments have shown that stack allocation is most important for relatively modest amounts of data—stack allocation for small data objects can decrease unmarshaling time by 14%—and that reuse of marshal buffer space is most important when the amount of data is large. However, the behavior of stack and marshal buffer storage means that it is suitable only in certain cases. Flick can identify these cases because it has access to the behavioral properties of the presentations that it creates.

2.3.2 Efficient Copying and Presentation

2.3.2.1 Data Copying

By comparing the encoded representation of an array with the representation that must be presented to or by a stub, Flick determines when it is possible to copy arrays of atomic types with the C function `memcpy`. Copying an object with `memcpy` is often faster than copying the same object component-by-component, especially when the components are not the same size as machine words. For instance, our measurements show that this technique can reduce character string processing times by 60–70%. In order for this optimization to be valid, the encoded and target language data formats must be identical, and this can be determined by examining the type chains constructed by the presentation generator and back end as described in Section 2.2.3. A more flexible copy optimizer that allows for byte swapping and word copying of other aggregate types—similar to the optimizer in USC [19]—will be implemented in a future version of Flick.

Even when an object cannot be copied with `memcpy`, Flick performs an optimization to speed component-by-component copying. As part of the analysis performed for optimizing marshal buffer allocation described above, Flick identifies portions of the message that have fixed layouts. A message region with a fixed size and a fixed internal layout is called a *chunk*. If Flick discovers that a stub must copy data into or out of a chunk, Flick produces code to set a *chunk pointer* to the address of the chunk. Subsequent stub accesses to components of the chunk are performed by adding a constant offset to the chunk pointer. The chunk pointer itself is not modified; rather, individual statements perform pointer arithmetic to read or write data. Flick assumes that the target language compiler will turn these statements into efficient pointer-plus-offset instructions. Chunking is a kind of common subexpression elimination that would not ordinarily be performed by the target language compiler itself due to the general difficulty of optimizing pointer-based code. Chunk-based code is more efficient than code produced by traditional IDL compilers, which generally increments a read or write pointer after each atomic datum is processed. Our experiments with Flick-generated stubs show that chunking can reduce some data marshaling times by 14%.

2.3.2.2 Specialized Transports

Because Flick is a toolkit, it is straightforward to implement back ends that take advantage of special features of a particular transport system. For example, Flick's Mach 3 back end allows stubs to communicate out-of-band data [20] and Flick's Fluke [10] back end produces stubs that communicate data between clients and servers in machine registers. A Fluke client stub stores the first several words of the message in a particular set of registers; small messages fit completely within the register set. When the client invokes the Fluke kernel to send the message, the kernel is careful to leave the registers intact as it transfers control to the receiving server. This optimization is critical for high-speed communication within many microkernel-based systems.

2.3.3 Efficient Control Flow

2.3.3.1 Inline Code

The stubs produced by many IDL compilers are inefficient because they invoke separate functions to marshal or unmarshal each datum in a message. Those functions in turn may invoke other functions, until ultimately, functions to process atomic data are reached. This type of code is straightforward for an IDL compiler to generate. However, these chains of function calls are expensive and impose a significant runtime overhead. Not only are the function calls wasteful, but reliance on separate, type-specific marshal and unmarshal functions makes it difficult for an IDL compiler to implement memory management optimizations such as those described previously in Section 2.3.1. A general-purpose marshal function must always check that buffer space is available; a separate unmarshal function cannot use the runtime stack to allocate space for the unmarshaled representation of a data object. Therefore, Flick aggressively inlines both marshal and unmarshal code into both client- and server-side stubs. In general, Flick-generated stubs invoke separate marshal or unmarshal functions only when they must handle recursive types such as linked lists or unions in which one of the union branches leads back to the union type itself.[5] For a large class of interfaces, inlining actually *decreases* the sizes of the stubs once they are compiled to machine code. This effect is illustrated in Table 2.2. Inlining obviously removes expensive function calls from the generated code, but more importantly, it allows Flick to specialize the inlined code *in context*. The memory, parameter, and copy optimizations described previously become more powerful as more code can be inlined and specialized. In total, our experiments with Flick show that stubs with inlined code can process complex data up to 60% faster than stubs without this optimization.

2.3.3.2 Message Demultiplexing

A server dispatch function must demultiplex messages received by the server process and forward those messages to the appropriate work functions. To perform this task, the dispatch function examines a discriminator value at the beginning of every message. This discriminator may be one or more integer values, a packed character string, or any other complex type, depending on the message format. Regardless of the type, Flick generates demultiplexing code that examines machine word-size chunks of the discriminator insofar as possible. The acceptable values for a discriminator word are used to produce a C `switch` statement; multi-word discriminators are decoded using nested `switch`es. When a

[5]A future version of Flick will produce iterative marshal and unmarshal code for "tail-recursive" data encodings in the marshal buffer.

Table 2.2. Object code sizes in bytes. Each IDL compiler produced stubs for the directory interface described in Section 2.4 and the generated stubs were compiled for our SPARC test machines. The sizes of the compiled stubs, along with the sizes of the library code required to marshal and unmarshal data, were determined through examination of the object files. Numbers for MIG are not shown because the MIG IDL cannot express the interface. Library code for ORBeline is not shown because we had limited access to the ORBeline runtime.

	Size of Client		**Size of Server**	
Compiler	**Stubs**	**Library**	**Stub**	**Library**
Flick	2800	0	2116	0
PowerRPC	2656	2976	2992	2976
`rpcgen`	2824	2976	3796	2976
ILU	7148	24032	6628	24032
ORBeline	14756		16208	

complete discriminator has been matched, the code to unmarshal the rest of the message is then inlined into the server dispatch function.

2.4 Experimental Results

To evaluate the impact of Flick's optimizations, we compared Flick-generated stubs to those from five other IDL compilers, including three sold commercially. The different IDL compilers are summarized in Table 2.3. The first compiler, Sun's `rpcgen`, is in widespread use. PowerRPC [17] is a new commercial compiler derived from `rpcgen`. PowerRPC provides an IDL that is similar to the CORBA IDL; however, PowerRPC's back end produces stubs that are compatible with those produced by `rpcgen`. ORBeline is a CORBA IDL compiler distributed by Visigenic, implementing the standard mapping for CORBA onto C++. ILU and MIG represent opposite ends of a spectrum: ILU is a very flexible compiler that produces slow stubs, whereas MIG is a very rigid compiler that produces fast stubs.

For the ONC RPC and CORBA IDL-based compilers, we measured the performance of generated stub functions communicating across three different networks: a 10 Mbps Ethernet link, a 100 Mbps Ethernet link, and a 640 Mbps Myrinet link [4]. For MIG interfaces, we measured Mach IPC speeds between separate tasks running on a single host.[6] For each transport and compiler, we measured the costs of three different method

[6]Our hosts for the network and marshaling tests were two Sun SPARCstation 20/50 machines. Each ran at

Table 2.3. Tested IDL compilers and their attributes. `rpcgen`, PowerRPC, and ORBeline are commercial products, while ILU and MIG are well known compilers from research organizations. The PowerRPC IDL is similar to the CORBA IDL. The target language was C, except for ORBeline which supports only C++.

Compiler	**Origin**	**IDL**	**Encoding**	**Transport**
`rpcgen`	Sun	ONC	XDR	ONC/TCP
PowerRPC	Netbula	~CORBA	XDR	ONC/TCP
Flick	Utah	ONC	XDR	ONC/TCP
ORBeline	Visigenic	CORBA	IIOP	TCP
ILU	Xerox PARC	CORBA	IIOP	TCP
Flick	Utah	CORBA	IIOP	TCP
MIG	CMU	MIG	Mach 3	Mach 3
Flick	Utah	ONC	Mach 3	Mach 3

invocations. The first method takes an input array of integers. The second takes an input array of "rectangle" structures: each structure contains two substructures, and each substructure holds two integers (i.e., a coordinate value). The third method takes an input array of variable-size "directory entry" structures: each directory entry contains a variable-length string followed by a fixed-size, UNIX `stat`-like structure containing 136 bytes of file information (30 4-byte integers and one 16-byte character array). Although the size of a directory entry is variable, in our tests we always sent directory entries containing exactly 256 bytes of encoded data.

These methods were repeatedly invoked in order to measure both marshaling speed and end-to-end throughput for a variety of message sizes. The first two methods were invoked to send arrays ranging in size from 64 bytes to 4MB. The third method was invoked to send arrays ranging in size from 256 bytes to 512KB.

2.4.1 Marshal Throughput

Marshal throughput is a measure of the time required for a stub to encode a message for transport, independent of other runtime overhead or the time required to actually transmit the message. To measure marshal throughput, we instrumented the stubs produced by Flick, `rpcgen`, PowerRPC, ILU, and ORBeline, and the resultant throughput measurements are shown in Figure 2.3. The figure shows that Flick-generated marshal code is between 2 and 5 times faster for small messages and between 5 and 17 times faster for large messages. As expected, Flick-generated stubs process integer arrays more quickly than structure arrays because Flick performs its `memcpy` optimization only for arrays of atomic types. ORBeline stubs use scatter/gather I/O in order to transmit arrays of integers and thereby avoid conventional marshaling [12]; this is why data for ORBeline's performance over integer arrays are missing from Figure 2.3.

50 MHz, had 20K/16K (I/D, 5/4 set-associative) L1 caches, no L2 caches, were rated 77 on the SPECint_92 benchmark, and had measured memory copy/read/write bandwidths of 35/80/62 MBps, although the `libc` `bcopy` gives only 29 MBps. They ran Solaris 2.5.1. One machine had 64 MB DRAM, while the other had 96 MB DRAM. Our host for the MIG tests was a 100 MHz Pentium with an 8K/8K (I/D 2/2 assoc) L1 cache, a 512 K direct-mapped L2 cache, both write-back, and 16 MB of DRAM, running CMU Mach 3. It had copy/read/write bandwidths of 36/62/82 MBps. All memory bandwidth tests were performed using `lmbench` 1.1 [16], and all throughput measurements were performed with the operating system socket queue size set to 64K.

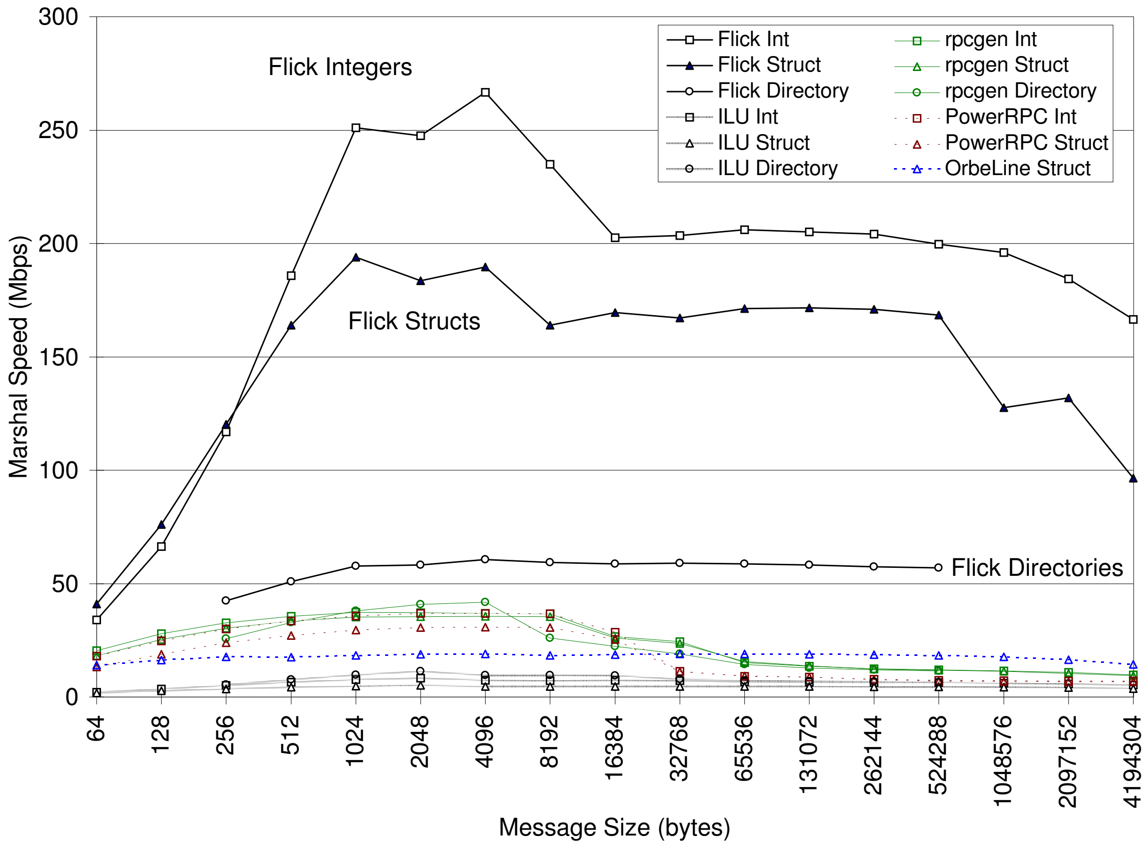


Figure 2.3. Marshal throughput on a big-endian (SPARC) architecture. This test compares equivalent marshaling functions and avoids any transport-related bottlenecks. The performance ratios are similar when the tests are performed on a little-endian (Pentium) architecture. Flick's superior throughput shows that Flick-generated stubs are suitable for use on high-performance transports.

2.4.2 End-to-End Throughput

The gains derived from greater marshal throughput can only be realized to the extent that the operating system and network between client and server do not limit the possible end-to-end throughput of the system. To show that improved network performance will increase the impact of an optimizing IDL compiler, we measured the round-trip performance of stubs produced by the three compilers supporting ONC transports: `rpcgen`, PowerRPC, and Flick, on three different networks. The stubs produced by the three compilers all have minimal runtime overhead that is not related to marshaling, thus allowing a fair comparison of end-to-end throughput.[7] Figure 2.4 shows that the maximum end-to-end throughput of all the compilers' stubs is approximately 6.5–7.5 Mbps when communicating across a 10 Mbps Ethernet. Flick's optimizations have relatively little impact on overall throughput.

Over fast communication links, however, Flick's optimizations again become very significant. Figures 2.5 and 2.6 show that for stubs communicating across 100 Mbps Ethernet and 640 Mbps Myrinet, Flick's optimizations increase end-to-end throughput by factors of 2–3 for medium size messages, factors of 3.2 for large Ethernet messages, and factors of 3.7 for large Myrinet messages. With Flick stubs, both 100 Mbps and 640 Mbps transports yield significant throughput increases. In contrast, PowerRPC and `rpcgen` stubs did not benefit from the faster Myrinet link: their throughput was essentially unchanged across the two fast networks. This indicates that the bottleneck for PowerRPC and `rpcgen` stubs is poor marshaling and unmarshaling behavior.

Measurements show that the principal bottlenecks for Flick stubs are the memory bandwidth of the SPARC test machines and the operating system's communication protocol stack. Flick's maximum throughput is less than half of the theoretical Ethernet bandwidth and less than 10% of the theoretical Myrinet bandwidth. These results, however, must be viewed in terms of the *effective* bandwidth that is available after memory and operating system overheads are imposed. As measured by the widely available `ttcp` benchmark program, the maximum effective bandwidth of our 100 Mbps Ethernet link is 70 Mbps and the maximum bandwidth of our Myrinet link is just 84.5 Mbps. These low measurements are due to the performance limitations imposed by the operating system's low-level

[7]Unlike stubs produced by Flick, `rpcgen`, and PowerRPC, stubs generated by ORBeline and ILU include function calls to significant runtime layers. These runtime layers perform tasks that are necessary in certain environments (e.g., multi-thread synchronization) but which are not required for basic client/server communication.

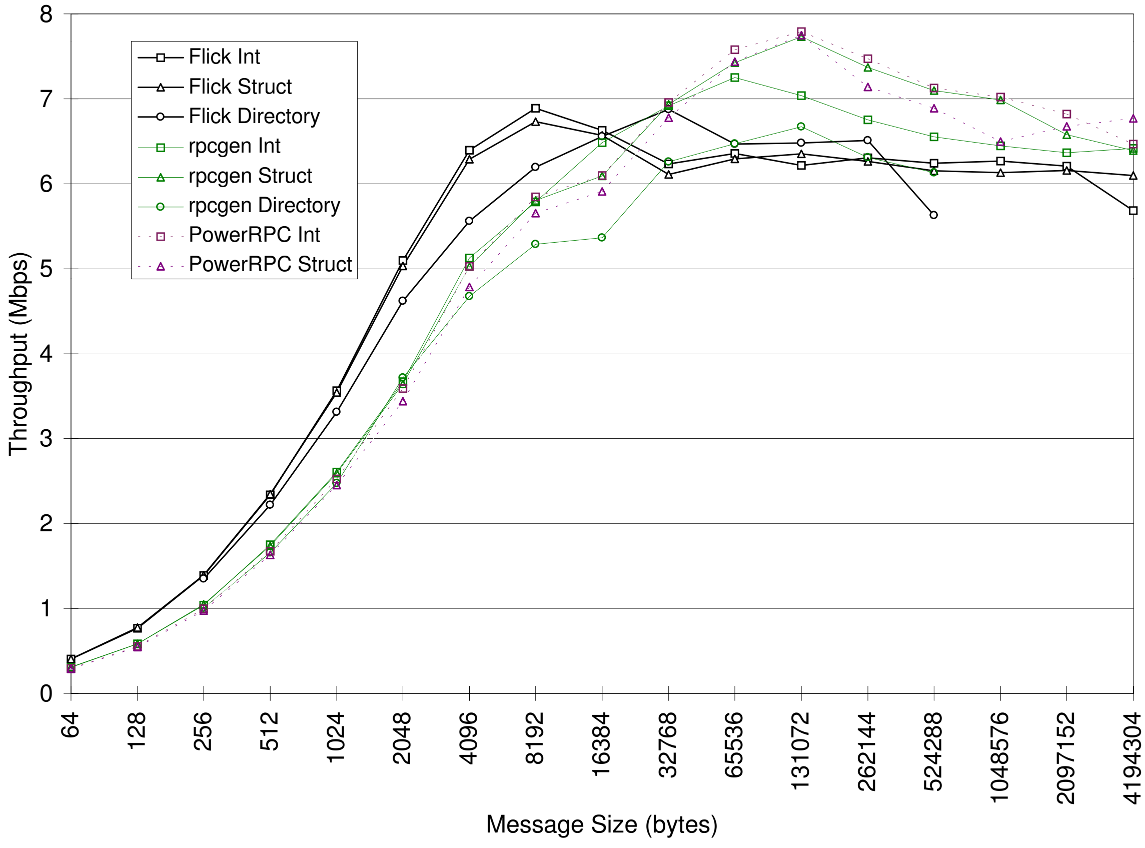


Figure 2.4. End-to-end throughput across 10 Mbps Ethernet. The data for PowerRPC and `rpcgen` is incomplete because the generated stubs signal an error when invoked to marshal large arrays of integers.

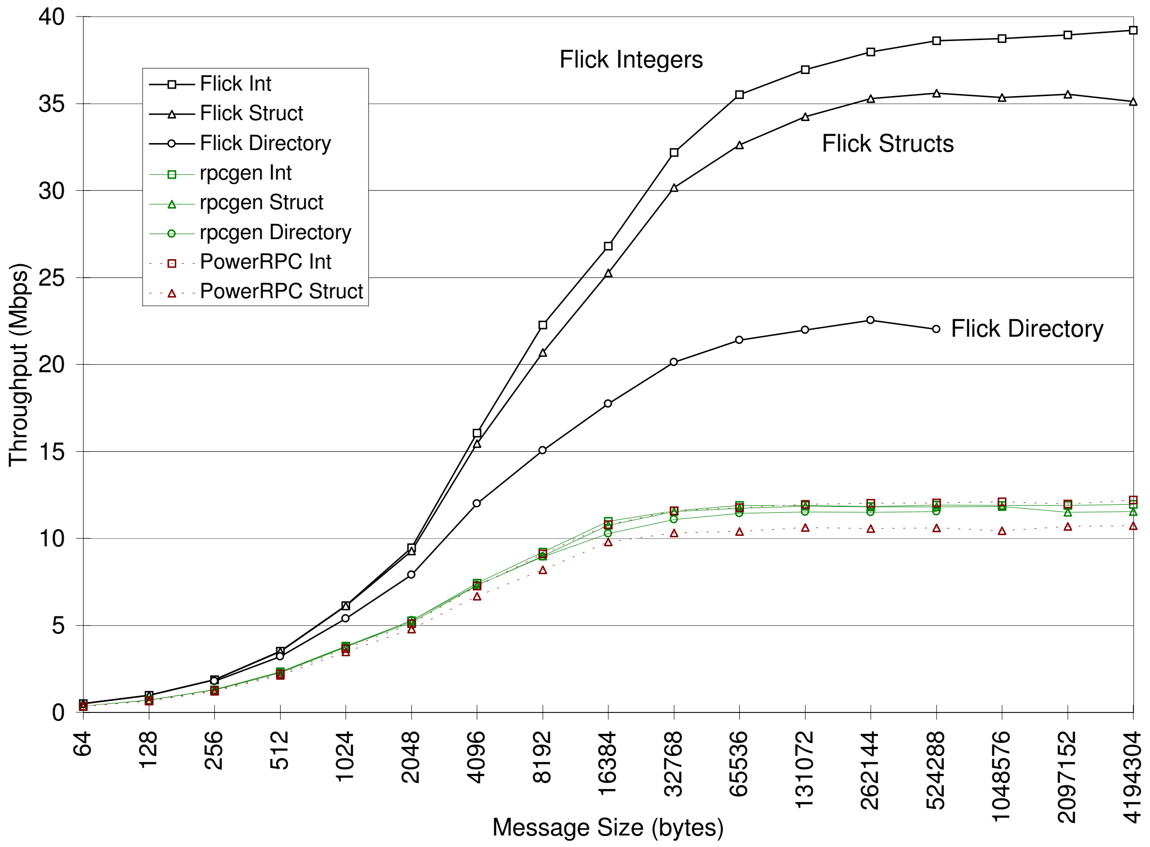


Figure 2.5. End-to-end throughput across 100 Mbps Ethernet

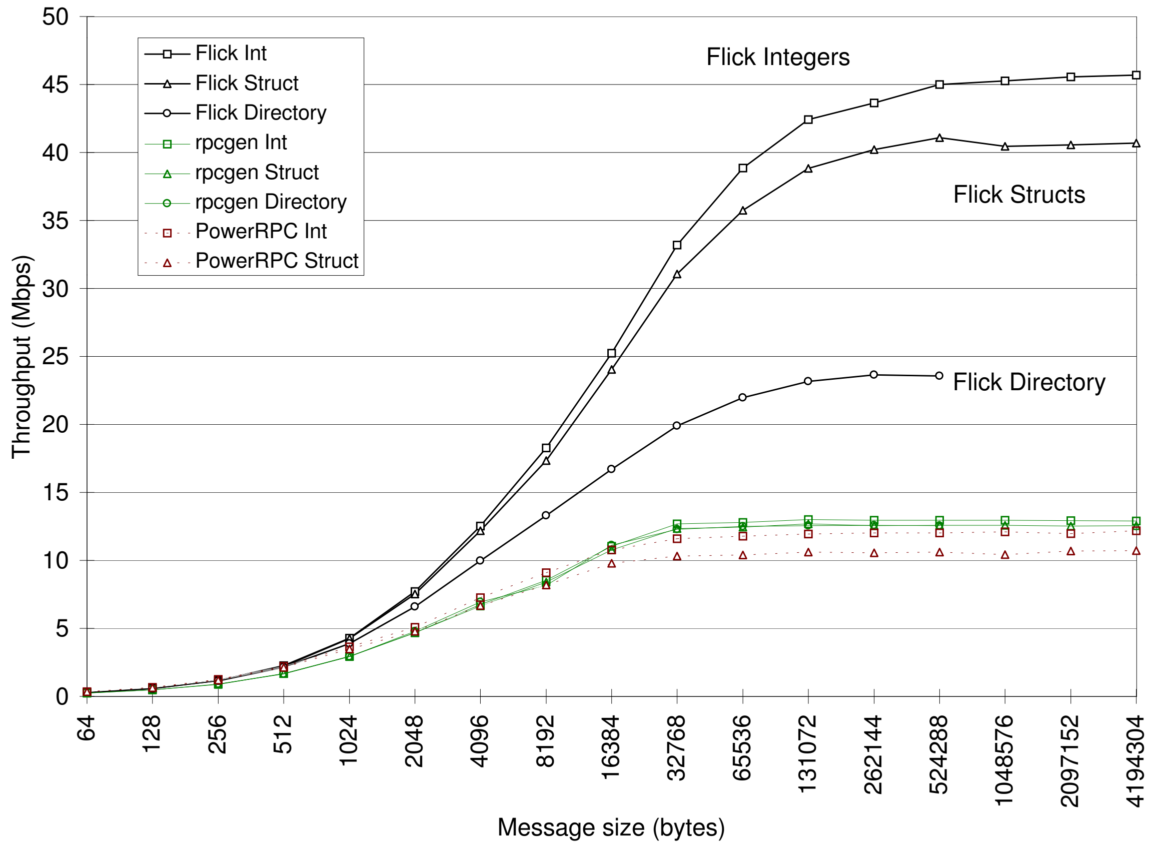


Figure 2.6. End-to-end throughput across 640 Mbps Myrinet

protocol layers [12]. Through calculations based on these numbers and measured memory bandwidth, we have confirmed that the difference between `ttcp` throughput and the performance of Flick stubs is entirely due to the functional requirement to marshal and unmarshal message data—which requires memory-to-memory copying and is thus limited by memory bandwidth. As operating system limitations are reduced by lighter-weight transports [6, 7], Flick's ability to produce optimized marshal code will have an increasingly large impact.

2.4.3 End-to-End Throughput Compared to MIG

In Figure 2.7 we compare the end-to-end throughput of Flick-generated stubs to the throughput of stubs generated by MIG, Mach 3's native IDL compiler. In this

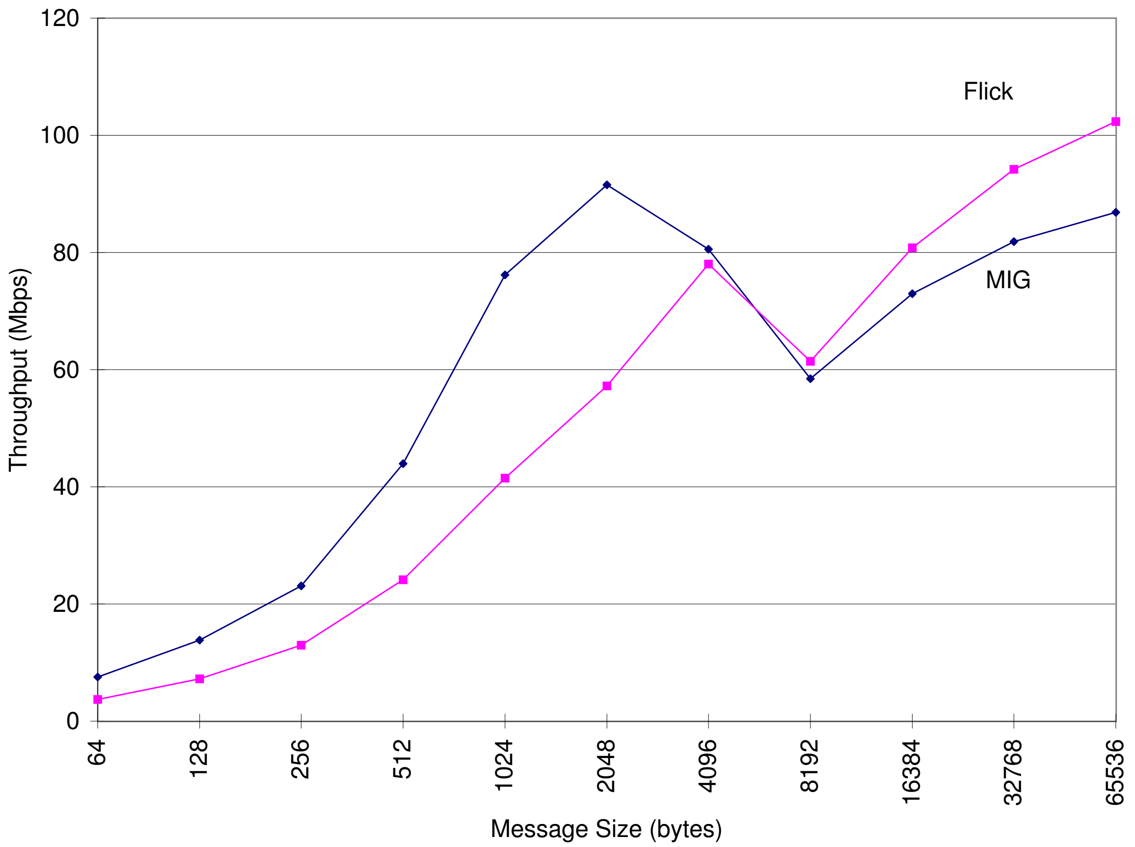


Figure 2.7. End-to-end throughput for MIG and Flick stubs. MIG is both highly specialized for optimizing Mach message communication, and is able only to support simple data types. At larger-sized messages, Flick-generated stubs achieve throughput comparable to that of MIG-generated stubs even though Flick is a much more flexible IDL compiler.

experiment the stubs transmit arrays of integers; we did not generate stubs to transmit arrays of structures because MIG cannot express arrays of non-atomic types. MIG is a highly restrictive IDL compiler, but it is also highly specialized for the Mach 3 message communication facility. The effect of this specialization is that for small messages, MIG-generated stubs have throughput that is twice that of the corresponding Flick stubs. However, as the message size increases, Flick-generated stubs do increasingly well against MIG stubs. Beginning with 8 KB messages, Flick's stubs increasingly outperform MIG's stubs, showing 17% improvement at 64 KB. The results of this experiment demonstrate the potential for further improvements in Flick and are encouraging because they show that although Flick is much more flexible and general-purpose than MIG, Flick-generated stubs can compete against stubs produced by the operating system's own IDL compiler. At a current cost for small and moderate sized messages, Flick allows Mach programmers to use modern IDLs such as CORBA and supports many C language presentations (e.g., structures) that MIG cannot offer.

2.5 Related Work

Previous work has shown that flexible, optimizing compilers are required in order to eliminate the crippling communication overheads that are incurred by many distributed systems. In a seminal paper in the networking domain, Clark and Tennenhouse [5] identified *data representation conversion* as a bottleneck to many communication protocols. They emphasized the importance of optimizing the presentation layer of a protocol stack and showed that it often dominates processing time. Recent work by Schmidt et al. [12, 21] has quantified this problem for `rpcgen` and two commercial CORBA implementations. On average, due to inefficiencies at the presentation and transport layers, compiler-generated stubs achieved only 16–80% of the throughput of hand-coded stubs.

To address these and similar performance issues, several attempts have been made to improve the code generated by IDL compilers. Mach's MIG [20] compiler generates fast code but only by restricting the types that it can handle: essentially just scalars and arrays of scalars. Hoschka and Huitema [14] studied the trade-offs between (large, fast) compiled stubs and (small, slow) interpreted stubs and suggested that an optimizing IDL compiler should use both techniques in order to balance the competing demands of throughput and stub code size. However, their experimental results appear to apply only to the extraordinarily expensive type representations used in ASN.1, in which type encoding is dynamic even for fundamental scalar types such as integers. Of more relevance to commonly used representations is the Universal Stub Compiler (USC) work by O'Malley et al. [19]. USC does an excellent job of optimizing copying based on a user-provided specification of the byte-level representations of data types. This work is complementary to ours: as the authors state, USC may be used alone to specify simple conversion functions (e.g., for network packet headers) or it may be leveraged by a higher-level IDL compiler. By incorporating USC-style representations for all types, Flick could improve its existing copy optimizations as outlined in Section 2.3.2.

Recently, Gokhale and Schmidt [13] addressed performance issues by optimizing SunSoft's reference implementation of IIOP [26]. The SunSoft IIOP implementation does not include an IDL compiler but instead relies on an interpreter to marshal and unmarshal data. The authors optimized the interpreter and thereby increased throughput over an ATM network by factors of 1.8 to 5 for a range of data types. Their implementation achieved throughput comparable to that of commercial CORBA systems that utilize compiled stubs, including ORBeline [11]. However, since Flick-generated stubs typically greatly outperform stubs produced by ORBeline, Flick must also outperform the best current interpretive marshalers.

In the area of flexible IDL compilers, the Inter-Language Unification [15] (ILU) system from Xerox PARC emphasizes support for many target languages, supporting C, C++, Modula-3, Python, and Common Lisp. However, like most IDL compilers, ILU uses as its sole intermediate representation a simple AST directly derived from the IDL input file. ILU does not attempt to do any optimization but merely traverses the AST, emitting marshal statements for each datum, which are typically (expensive) calls to type-specific marshaling functions. Each separate backend is essentially a full copy of another with only the `printf`s changed. For Flick to do similarly, it would simply emit marshaling code as it traversed an AOI structure. ILU does support two IDLs—its native, unique IDL and the CORBA IDL—but only by translating the CORBA language into its own IDL.

Like Flick, the Concert/C distributed programming system [2] quite fully develops the concept of flexible presentation. In Concert, the primary purpose of this separation is to handle the vagaries of RPC mapping to different target languages, striving for a "minimal contract" in order to achieve maximal interoperability between target languages. However, this separation is not leveraged for optimizations. In earlier work [8, 9] we concentrated on leveraging Flick's explicit separation of presentation from interface in order to produce application-specialized stubs. We showed that programmer-supplied interface annotations that coerce the “programmer’s contract” to applications’ needs could provide up to an order of magnitude speedup in RPC performance.

Finally, several techniques used by Flick are similar or analogous to those in traditional compilers for general purpose programming languages. In addition, it appears that our work has many similarities to type-based representation analysis [22] directed to achieving more efficient “unboxed” data representations whenever possible, and to convert between such representations.

2.6 Conclusion

This work exploits the fundamental and overdue recognition that interface definition languages are indeed programming languages, albeit specialized and nontraditional in their computational content. This insight is the basis for Flick, a novel, modular, and flexible IDL compiler that approaches stub generation as a programming language translation problem. This, in turn, allows established optimizing compiler technology to be applied and extended in domain-specific ways.

Flick exploits many fundamental concepts of modern compiler organization including carefully designed intermediate representations, modularized front and back ends localizing source and target language specifics, and a framework organization that encourages reuse of software implementing common abstractions and functionality. Our quantitative experimental results confirm that this approach is indeed effective for producing high-performance stubs for a wide variety of communication infrastructures.

2.7 Availability

Complete Flick source code and documentation are available at `http://www.cs.utah.edu/projects/flux/flick/`.

2.8 Acknowledgments

We are especially indebted to Steve Clawson, who provided technical support for our Mach and Myrinet performance measurements. Chris Alfeld, Godmar Back, John Carter, Ajay Chitturi, Steve Clawson, Mike Hibler, Roland McGrath, Steve Smalley, Jeff Turner, and Kevin Van Maren all reviewed drafts of this paper and suggested numerous improvements; we wish we could have incorporated them all. Nathan Dykman and Gary Crum did much early implementation. We thank Gary Barbour for loaning us the Suns, Al Davis and

Chuck Seitz for loaning the Myrinet cards, and Grant Weiler for help with Sun system software.

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602–96–2–0269. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

2.9 References

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer 1986 USENIX Conf.* (June 1986), pp. 93–112.

[2] Auerbach, J. S., and Russell, J. R. The Concert signature representation: IDL as an intermediate language. In *Proc. of the Workshop on Interface Definition Languages* (Jan. 1994), pp. 1–12.

[3] Birrell, A. D., and Nelson, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984).

[4] Boden, N., Cohen, D., Felderman, R., Kulawik, A., Seitz, C., Seizovic, J., and Su, W.-K. Myrinet – A gigabit-per-second local-area network. *IEEE MICRO 15*, 1 (February 1995), 29–36.

[5] Clark, D. D., and Tennenhouse, D. L. Architectural considerations for a new generation of protocols. In *Proc. of the SIGCOMM '90 Symp.* (1990), pp. 200–208.

[6] Druschel, P., Davie, B. S., and Peterson, L. L. Experiences with a high-speed network adapter: A software perspective. In *Proc. of the SIGCOMM '94 Symp.* (1994), pp. 2–13.

[7] Eicken, T. v., Culler, D. E., Goldstein, S. C., and Schauser, K. E. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19th International Symp. on Computer Architecture* (May 1992), pp. 256–266.

[8] Ford, B., Hibler, M., and Lepreau, J. Using annotated interface definitions to optimize RPC. In *Proc. of the 15th ACM Symp. on Operating Systems Principles* (1995), p. 232. Poster.

[9] Ford, B., Hibler, M., and Lepreau, J. Using annotated interface definitions to optimize RPC. Tech. Rep. UUCS-95-014, University of Utah, Mar. 1995.

[10] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. Microkernels meet recursive virtual machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation* (Seattle, WA, Oct. 1996), USENIX Assoc., pp. 137–151.

[11] GOKHALE, A. Personal communication, Mar. 1997.

[12] GOKHALE, A., AND SCHMIDT, D. C. Measuring the performance of communication middleware on high-speed networks. *Computer Communication Review 26*, 4 (Oct. 1996).

[13] GOKHALE, A., AND SCHMIDT, D. C. Optimizing the performance of the CORBA Internet Inter-ORB Protocol over ATM. Tech. Rep. WUCS–97–09, Washington University Department of Computer Science, St. Louis, MO, 1997.

[14] HOSCHKA, P., AND HUITEMA, C. Automatic generation of optimized code for marshalling routines. In *International Working Conference on Upper Layer Protocols, Architectures and Applications* (Barcelona, Spain, 1994), M. Medina and N. Borenstein, Eds., IFIP TC6/WG6.5, North-Holland, pp. 131–146.

[15] JANSSEN, B., AND SPREITZER, M. *ILU 2.0alpha8 Reference Manual*. Xerox Corporation, May 1996. `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[16] MCVOY, L., AND STAELIN, C. lmbench: Portable tools for performance analysis. In *Proc. of 1996 USENIX Conf.* (Jan. 1996).

[17] NETBULA, LLC. PowerRPC, Version 1.0, 1996. `http://www.netbula.com/products/powerrpc/`.

[18] OBJECT MANAGEMENT GROUP. *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[19] O'MALLEY, S., PROEBSTING, T. A., AND MONTZ, A. B. USC: A universal stub compiler. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM)* (London, UK, Aug. 1994), pp. 295–306.

[20] OPEN SOFTWARE FOUNDATION AND CARNEGIE MELLON UNIVERSITY. *Mach 3 Server Writer's Guide*. Cambridge, MA, Jan. 1992.

[21] SCHMIDT, D. C., HARRISON, T., AND AL-SHAER, E. Object-oriented components for high-speed network programming. In *Proceedings of the First Conference on Object-Oriented Technologies and Systems* (Monterey, CA, June 1995), USENIX.

[22] SHAO, Z., AND APPEL, A. A type-based compiler for standard ML. In *Proc. ACM SIGPLAN Symp. on Programming Language Design and Implementation* (June 1995), pp. 116–129.

[23] SRINIVASAN, R. RPC: Remote procedure call protocol specification version 2. Tech. Rep. RFC 1831, Sun Microsystems, Inc., Aug. 1995.

[24] SRINIVASAN, R. XDR: External data representation standard. Tech. Rep. RFC 1832, Sun Microsystems, Inc., Aug. 1995.

[25] SUN MICROSYSTEMS, INC. *ONC+ Developer's Guide*, Nov. 1995.

[26] SUNSOFT, INC. SunSoft Inter-ORB Engine, Release 1.1, June 1995. `ftp://ftp.omg.org/pub/interop/iiop.tar.Z`.

PART II

STATIC AND DYNAMIC STRUCTURE IN DESIGN PATTERNS

Reprinted from *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 208–218, May 2002: “Static and Dynamic Structure in Design Patterns” by Eric Eide, Alastair Reid, John Regehr, and Jay Lepreau. © 2002 Association for Computing Machinery, Inc. Reprinted by permission. http://doi.acm.org/10.1145/581339.581367

CHAPTER 3

STATIC AND DYNAMIC STRUCTURE IN DESIGN PATTERNS

Design patterns are a valuable mechanism for emphasizing structure, capturing design expertise, and facilitating restructuring of software systems. Patterns are typically applied in the context of an object-oriented language and are implemented so that the pattern participants correspond to object instances that are created and connected at run-time. This paper describes a complementary realization of design patterns, in which many pattern participants correspond to statically instantiated and connected components.

Our approach separates the static parts of the software design from the dynamic parts of the system behavior. This separation makes the software design more amenable to analysis, thus enabling more effective and domain-specific detection of system design errors, prediction of run-time behavior, and more effective optimization. This technique is applicable to imperative, functional, and object-oriented languages: we have extended C, Scheme, and Java with our component model. In this paper, we illustrate our approach in the context of the OSKit, a collection of operating system components written in C.

3.1 Introduction

Design patterns allow people to understand computer software in terms of stylized relationships between program entities: a pattern identifies the roles of the participating entities, the responsibilities of each participant, and the reasons for the connections between them. Patterns are valuable during the initial development of a system because they help software architects outline and plan the static and dynamic structure of software before that structure is implemented. Documented patterns are useful for subsequent system maintenance and evolution because they help maintainers understand the software implementation in terms of well-understood, abstract structuring concepts and goals.

The conventional approach to realizing patterns [13] primarily uses classes and objects to implement participants and uses inheritance and object references to implement relationships between participants. The parts of patterns that are realized by classes and inheritance correspond to static information about the software—information that can be essential for understanding, checking, and optimizing a program. Unfortunately, class structures can disguise the underlying pattern relationships, both by being too specific (to a particular application of a pattern) and by being mixed with unrelated code. In contrast, the parts of patterns realized by run-time objects and references are more dynamic and flexible, but are therefore harder to understand and analyze.

This paper describes a complementary approach to realizing patterns based on separating the static parts of a pattern from the dynamic parts. The *static participants* and relationships in a pattern are realized by component instances and component interconnections that are set at compile- or link-time, while the *dynamic participants* continue to be realized by objects and object references. Expressing static pattern relationships as component interconnections provides more flexibility than the conventional approach while also promoting ease of understanding and analysis.

To illustrate the trade-offs between these approaches, consider writing a network stack consisting of a TCP layer, an IP layer, an Ethernet layer, and so on. The usual implementation strategy, used in mainstream operating systems, is for the implementation of each layer to directly refer to the layers above and below except in cases where the demand for diversity is well understood (e.g., to support different network interface cards). This approach commits to a particular network stack when the layers are being written, making it hard to change decisions later (e.g., to add low-level packet filtering in order to drop denial-of-service packets as early as possible).

An alternate implementation strategy is to implement the stack according to the *Decorator*[1] pattern with objects: each layer is implemented by an object that invokes methods in objects directly above and below it. The objects at each layer provide a common interface (e.g., methods for making and breaking connections, and for sending and receiving packets), allowing the designer to build a large variety of network stacks. In fact, stacks can be reconfigured at run-time, but that is more flexibility than most users require.

Our design and implementation approach offers a middle ground. Having identified the *Decorator* pattern and having decided that the network stack may need to be recon-

[1]Unless otherwise noted, the names of specific patterns refer to those presented in Gamma et al.'s *Design Patterns* catalog [13].

figured, but not at run-time, each decorator would be implemented as a component that imports an interface for sending and receiving packets and exports the same interface. The choice of network stack is then statically expressed by connecting a particular set of components together.

The basis of our approach is to permit system configuration and realization of design patterns at *compile-* and *link-time* (i.e., before software is deployed) rather than at *init-* and *run-time* (i.e., after it is deployed). Components are defined and connected in a language that is separate from the base language of our software, thus allowing us to separate "configuration concerns" from the implementation of a system's parts. A system can be reconfigured at the level of components, possibly by a nonexpert, and can be analyzed to check design rules or optimize the overall system. Our approach helps the programmer identify design trade-offs and strike an appropriate balance between design-time and run-time flexibility.

The contributions of this paper are as follows:

- We describe an approach to realizing patterns that clearly separates the static parts of the design from the dynamic parts, making the system more amenable to optimization and to analyses that detect errors or predict run-time behavior (Section 3.3).
- We define a systematic method for applying our approach to existing patterns (Section 3.3.1).
- We show that our approach is applicable to three major programming language paradigms that support the unit component model: imperative languages, exemplified by C [21]; functional languages, exemplified by Scheme [11]; and object-oriented languages, exemplified by Java [17] (Sections 3.2 and 3.3). We demonstrate our approach with two examples from the OSKit [12], a set of operating system components written in C (Sections 3.3.2 and 3.3.3).
- We evaluate the approach by applying it to each pattern described by Gamma et al. [13] (Section 3.3.4) and by analyzing its costs and benefits (Section 3.4).

In summary, although the benefits of separating system architecture from component implementations are well-known, the distinctive features of this paper are that: we show a general approach that can be applied to many patterns and in multiple language paradigms; we consider the static-dynamic decision in the context of design patterns; and we thoroughly evaluate when to apply and when not to apply our approach.

3.2 The Unit Model

Our approach to realizing patterns is most readily expressed in terms of *units* [10, 11], a component definition and linking model in the spirit of the Modula-3 and Mesa [19] module systems. The unit model emphasizes the notion of components as reusable architectural elements with well-defined interfaces and dependencies. It fits well with the definitions of "component" in the literature [22, p. 34] but differs from other component models that emphasize concerns such as separate compilation and dynamic component assembly. In the unit model, components are compile- or link-time parts of an assembly: i.e., software modules, not run-time objects.

Three separate implementations of the unit model exist: *Knit* [21] for C, *Jiazzi* [17] for Java, and *MzScheme* [11] for Scheme. The implementations differ in details both because of technical differences in the base languages and because of stylistic differences in the way the base languages are used. For the purposes of this paper, we focus on the common features of the three implementations.

3.2.1 Atomic and Compound Units

An *atomic unit* can be thought of as a module with three parts: (1) a set of *imports* that name the dependencies of the unit, i.e., the definitions that the unit requires; (2) a set of *exports* that name the definitions that are provided by the unit and made available to other units; and (3) an implementation, which must include a definition for each export, and which may use any of the imports as required.

Each import and export is a *port* with a well-defined *interface*. An interface has a name and serves to group related terms, much like an interface or abstract class in an OOP language. The three implementations of the unit model make different choices about what makes up an interface. In Knit, an interface refers to sets of related C types, function prototypes, and variable declarations. In Jiazzi, port interfaces are like Java packages: they describe partial class hierarchies and the public methods and fields of classes. In MzScheme, because Scheme uses run-time typing, interfaces are simply lists of function names.

Definitions that are not exported are inaccessible from outside the unit. The implementation of a unit is usually stored in a file separate from the unit definition, allowing code that was not intended for use as a unit to be packaged up as a unit.

Although all implementations of the unit model use a textual language to define units, in this paper we use a graphical notation to avoid inessential details and to emphasize the underlying structure of our systems. The smaller boxes in Figure 3.1 represent atomic units. The export interfaces are listed at the top of a unit, the import interfaces are listed at the bottom, and the name of the unit is shown in the center. Consider the topmost unit, called Draw. It has the ability to load, save, and render images, encapsulating the main parts of a simple image viewing program. Draw exports (i.e., implements) one port with interface I_Main and imports two ports: one with interface I_Widget and a second with interface I_File.

Units are instantiated and interconnected in *compound units*. Like atomic units, compound units have a set of imports and a set of exports that define connection points to other units. The implementation of a compound unit consists of a set of unit instances and a set of explicit interconnections between ports on these instances and the imports and exports of the compound unit. The result of composing units is a new unit, which is available for further linking.

Figure 3.1 as a whole represents a compound unit composed of three other units. In this figure, an instance of Draw is composed with an instance of Win32 Widgets and an

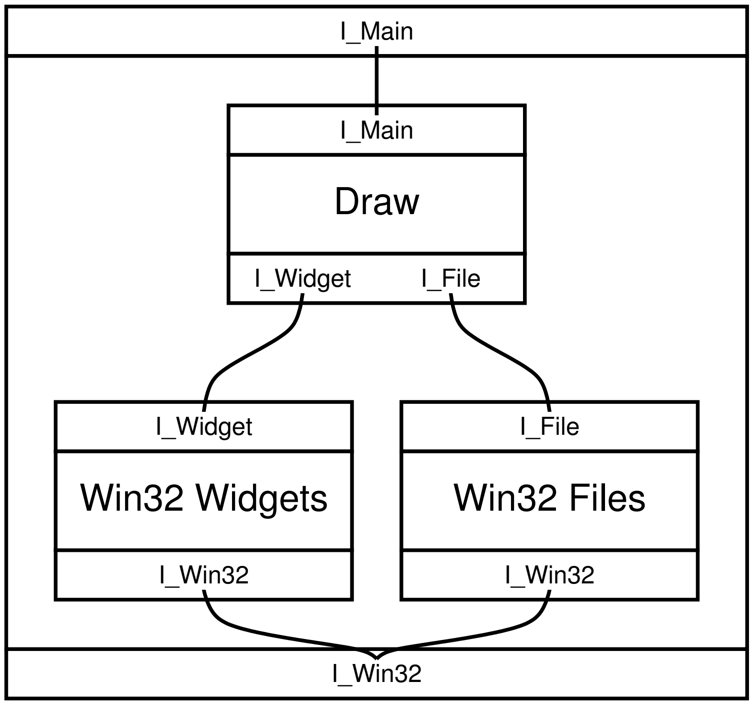


Figure 3.1. Atomic and compound units

instance of Win32 Files. Within a compound unit, connections are defined *explicitly*: this is necessary when there is more that one way to connect the units. Although not shown in this example, a system designer may freely create multiple unit instances from a single unit definition (e.g., two instances of Draw).

3.2.2 Exploiting Static Configuration

One of the key properties of programming with the unit component model is that component instantiation and interconnection are performed when the program is built instead of when the program is executed. This allows implementations of the unit model to make use of additional resources that may be available at compile- and link-time: powerful analysis and optimization techniques; in the case of embedded systems, orders of magnitude more cycles and memory with which to perform analyses; test cases, test scaffolding, and debugging builds; and finally, freedom from real-world constraints such as real-time deadlines. All three unit implementations check the component composition for type errors. Knit, which implements units for C, provides additional features that exploit the static nature of unit compositions.

3.2.2.1 Constraint Checking

Even if every link in a unit composition is "correct" according to local constraints such as type safety, the system *as a whole* may be incorrect because it does not meet global constraints. For example, [21] describes a design constraint used by operating system designers: "bottom-half code," executed by interrupt handlers, must not invoke "top-half code" that executes in the context of a particular process. The reason is that while top-half code typically blocks when a resource is temporarily unavailable, storing its state in the process's stack, an interrupt handler lacks a process context and therefore must not block. The problem with enforcing this constraint is that units containing bottom-half code (e.g., device drivers) may invoke code from other units that, transitively, invokes a top-half unit. Keeping track of such conditions is difficult, especially when working with low-level systems code that is highly interconnected and not strictly layered. To address this problem, Knit unit definitions can include constraint annotations that describe the properties of imports and exports. Constraints can be declared explicitly (e.g., that imported functions are invoked by bottom-half code) or by description (e.g., that the import properties are set by the exports). At system build-time, Knit propagates unit properties in order to ensure that all constraints are satisfied.

3.2.2.2 Cross-Component Inlining

When source is available, Knit inlines function definitions across component boundaries with the help of the C compiler. By eliminating most of the overhead associated with componentization, Knit reduces the need to choose between a clean design and a fast implementation.

3.2.3 Using Units Without Language Support

The unit model makes it possible for a software architect to design a system from components, describe local and global relationships between components, and reuse components both within and across system designs. These are the features that make it useful to develop and apply units for expressing design patterns. In particular, our unit-based approach to realizing patterns relies on these features of the unit model:

- **Programming to interfaces.** The only connections between components are through well-typed interfaces.
- **Configurable intercomponent connections.** Unit imports describe the "shapes" but not the providers of required services. A system architect links unit instances as part of a system definition, not as part of a component's base (e.g., C or Java) implementation.
- **Static component instantiation and interconnection.** Units are instantiated and linked when the system is built, not when the system is run.
- **Multiple instantiation.** A single unit definition can be used to create multiple unit instances, each of which has a unique identity at system build-time. Each instance can be linked differently.

It is possible to make use of features of the unit component model without support from languages such as Knit, Jiazzi, and MzScheme. However, without support, some benefits of the model may be lost. For instance, a C++ programmer might use template classes to describe units: this can provide optimization benefits but does not help the system designer check constraints of the sort described previously. A C programmer might use the C preprocessor to achieve similar results. In sum, although unit tools can provide important benefits, people who cannot or decide not to use our unit description languages can nevertheless take advantage of our general approach to realizing design patterns.

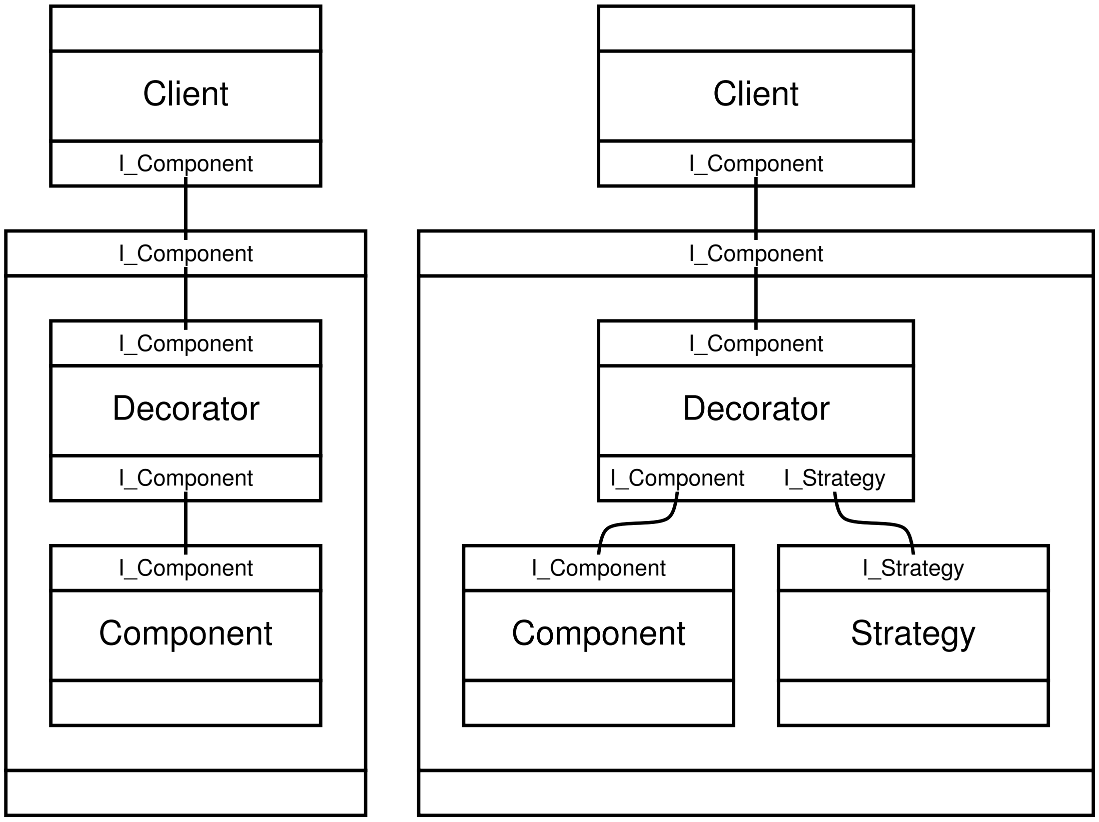
3.3 Expressing Patterns with Units

The essence of a design pattern is the set of participants in the pattern and the relationships between those participants. As outlined previously, the conventional approaches to describing and realizing patterns are based on the idioms of object-oriented programming. At design-time, the participants in the pattern correspond to classes. At run-time, the pattern is realized by object instances that are created, initialized, and connected by explicit statements in the program code. This style of implementation allows for a great deal of run-time flexibility, but in some cases it can disguise information about the static properties of a system—information that can be used to check, reason about, or optimize the overall system.

The key idea of this paper is that it is both possible and fruitful to separate static knowledge about a pattern application from dynamic knowledge. In particular, we believe that static information should be "lifted out" of the ordinary source code of the system, and should be represented at the level of unit definitions and connections. The unit model allows a system architect to describe the static properties of a system in a clear manner, and to separate "configuration concerns" from the implementations of the system's parts.

Consider, for example, an application of the *Decorator* pattern: this pattern allows a designer to add additional responsibilities to an entity (e.g., component or object) in a way that is transparent to the clients of that entity. One might apply *Decorator* to protect a non-thread-safe singleton component with a mutual exclusion wrapper (which acquires a lock on entering a component and releases the lock on exit) when using the component in a multithreaded environment. In an object-oriented setting, this pattern would often be realized by defining three classes: one abstract class to define the component interface, and two derived classes corresponding to the concrete component and decorator. At init-time, the program would create instances of each concrete class and establish the appropriate object connections. While workable, this implementation of the pattern can disguise valuable information about the static properties of this system. First, it hides the fact that there will be only one instance each of the component and decorator. Second and more important, it hides the design constraint that the base component must be accessed only through the decorator: because the realization of the pattern doesn't enforce the constraint, future changes to the program may violate the rule.

To overcome these problems, we would realize the *Decorator* pattern at the level of units, as illustrated in Figure 3.2(a). We create one unit definition to encapsulate the base



(a) Decorator (b) Strategized Decorator

Figure 3.2. Units realizing *Decorator* patterns

component definition; by instantiating this definition exactly once, we make it clear that there will be only one instance in the final program. Furthermore, we annotate the unit definition with the constraint that the implementation is non-thread-safe. We then create a separate unit definition to encapsulate our decorator, and include in the definition a specification that it imports a non-thread-safe interface and exports a thread-safe one. The resulting structure in Figure 3.2(a) makes it clear that there is one instance of each participant and that there is no access to the base component except through the decorator. Units make the static structure of the system clear, and unit compositions can be checked by tools to enforce design constraints. Of course, unit definitions are reusable between systems (and within a single system): we can include the decorator instances only as needed. If we desire greater reuse, we can apply the *Strategy* pattern to our decorator to separate its wrapping and locking aspects as shown in Figure 3.2(b). This structure provides greater flexibility while still allowing for cross-component reasoning and optimization when the strategy is statically known.

In sum, our approach to realizing patterns promotes the benefits of static knowledge within patterns by moving such information to the level of units. The unit model allows us to describe and separate the static and dynamic properties of a particular pattern application, thus making it possible for us to exploit the features described in Section 3.2.2. In the sections below we define a method for applying our approach, demonstrate the method in detail on a small example, demonstrate the effect of our method on a large example, and consider how the method applies to each of the patterns in Gamma et al.'s *Design Patterns* catalog [13].

3.3.1 A Method for Expressing Patterns with Units

In realizing a pattern via units, the software architect's task is to identify the parts of the pattern that correspond to static (compile-time or link-time) knowledge about the pattern and its participants, to "lift" that knowledge out of the implementation code, and then to translate that knowledge into parts of unit definitions and connections. This process is necessarily specific to individual uses of a pattern: each time a pattern is applied, the situation dictates whether certain parts of the pattern correspond to static or dynamic knowledge. In our experience, however, we have found that many patterns are commonly applied in situations that provide significant amounts of static information, and which therefore allow system architects to exploit the features of the unit model.

We have found the following general procedure to be useful in analyzing the application of a pattern and translating that pattern into unit definitions, instances, and linkages. Because patterns are ordinarily described in terms of object-oriented structures (classes, interfaces, and inheritance), we describe our method as a translation from object-oriented concepts to parts of the unit model.

1. Identify the abstract classes/interfaces. Many pattern descriptions contain one or more participating abstract classes that serve to define a common interface for a set of derived classes. The abstract classes therefore serve the same purpose as interfaces in the unit model; the three implementations of the model all allow related operations (and types, if needed) to be grouped together in named interfaces. In Figure 3.2(a), for example, I_Component corresponds to the abstract component class described in the *Decorator* pattern. The exact translation from abstract class to unit interface depends on whether or not the derived classes are "static participants" in the application of the pattern at hand, as described next.

2. Identify the "static" and "dynamic" participants within the pattern. Within the context of a pattern, it is often the case that some pattern participants will be realized by a small and statically known number of instances. For example, in uses of the *Abstract Factory* pattern (see Section 3.3.2), there will often be exactly one *Concrete Factory* instance in the final system (within the scope of the pattern). The number of instances does not need to be exactly one: what is important is that the number of instances, their classes, and the inter-instance references are all known statically.

We refer to these kinds of participants as *static participants*, and in the steps below, we realize each of these participants as an individual unit instance—essentially, realizing the participant as a part of our static architecture, rather than as a run-time object. In Figure 3.2(a), the context of our example says that in this particular use of *Decorator*, both the base component and its decorator are singletons. Thus, they are static participants.

If a pattern participant is not static we refer to it as a *dynamic participant*. In this case, we will translate the participant as a unit that will encapsulate the participant class and will be able to produce instances at run-time. Figure 3.2(a) has no dynamic participants; later examples will show their use.

3. Define the interfaces for static participants. Following the class hierarchy of the pattern, the software architect defines the unit interfaces to group the operations that will be provided by the static participants. The architect may choose to create one interface per class (i.e., one interface for the new operations provided at each level of the class inheritance hierarchy), or may group the operations at a finer granularity.

Because each instance of a static participant will be implemented by a unique unit instance in the realization of the pattern, the identity of each instance is part of the static system architecture and need not be represented by an object at run-time. Therefore, in the translation from class to unit interface, the methods that constitute the interface to a static participant can be translated as ordinary functions (or as class static methods, in the case of Jiazzi), and data members can be translated as ordinary variables (static members). Any method arguments that represent references to static participants can be dropped from the translated function signatures: these arguments will be replaced by imports to the unit instances (i.e., explicit, unit-level connections between the static participants).

Thus, in the running example of Figure 3.2(a), the operations in I_Component can be implemented by ordinary functions. Because all of our participants are static, we do not need to represent them as run-time objects.

4. Define the interfaces for dynamic participants. Following the class hierarchy of the pattern, the designer now creates the interfaces for the dynamic participants. As described for the previous step, the designer may choose to create one or several interface definitions per class.

Unlike the static case, each instance of a dynamic participant must be represented by a run-time object (or other entity in a non-OOP language). This means that in translating the participant class to unit interfaces, the designer must include the definition of the type of the run-time objects, as the implementation language requires. With Jiazzi, this is straightforward: Jiazzi unit interfaces contain Java class definitions. In Knit, the interface would include a C type name along with a set of functions, each of which takes an instance of that type as an argument (i.e., the "self" parameter). MzScheme is the simplest: because Scheme uses run-time typing, the unit interface does not need to include the type of the dynamic pattern participants at all.[2]

Although the interfaces for a dynamic participant must include the class of the participant objects, the unit model allows the designer to avoid hard-coding class inheritance knowledge into the interfaces. By writing our units so that they import the superclasses of each exported class, we can implement our dynamic participant classes in a manner

[2]If the pattern structure relies on implementation inheritance, dynamic method dispatch, or other essentially OOP features, these capabilities must be emulated when translating the pattern to Knit or MzScheme units. In our experience, this is sometimes tedious but generally not too difficult.

corresponding to *mixins* [8, 17]. In other words, we can represent the static inheritance relationships between pattern participants not in the definitions of our units or in the unit interfaces, but in the connections between units.

5. Create a unit definition for each concrete (static or dynamic) participant. With the interfaces now defined, the designer can write the definitions of the units for each participant. The unit definition for a dynamic participant encapsulates the *class* of the dynamic instances; normally, in the context of a single pattern, these kinds of units will be instantiated once. The unit definition for a static participant, on the other hand, encapsulates a single *instance* of the participant. The unit definition for a static participant may be instantiated as many times as needed, each time with a possibly different set of imports, to create all of the needed static participant instances. In either case, the exports of a unit correspond to the services that the participant provides. The imports of a unit correspond to the connections described in the pattern; the imports of each unit instance will be connected to the exports of other unit instances that represent the other (static and dynamic) participants.

Continuing the example of Figure 3.2(a), the software designer writes definitions for the Component and Decorator units, each encapsulating a single instance of the corresponding participant. The base component has an I_Component export, while the Decorator both imports and exports that interface.

6. Within a compound unit definition, instantiate and connect the participant units. Within a compound unit, the designer describes the instantiation of the pattern as a whole. The implementation of the compound unit specifies how the participant units are to be instantiated and connected to one another. The connections between units follow naturally from the structure of the pattern and its application in the current context. In addition, one must import services that are required by the encapsulated participants.

The above considers just one pattern applied before any code is written. In practice, participants have roles in multiple patterns and patterns are applied during code evolution. These considerations necessitate changes such as omitting the enclosing compound unit, moving some participants outside the compound unit, or choosing to treat a static participant as dynamic (or vice versa) to avoid extensive changes to the implementations of the participants. The system designer may want to make additional changes, such as aggregating groups of interfaces into single interfaces, to reduce the complexity of the unit descriptions.

3.3.2 Example: Managing Block Devices

We illustrate our approach in the context of a concrete system. The OSKit [12] is a collection of components for building operating systems and standalone systems. The components are almost all written in C, with a few in assembly code. Although the OSKit includes a number of small and modest-sized "from-scratch" components, such as memory and thread management, the majority of its code is taken from elsewhere, including FreeBSD, NetBSD, Linux, and the Mach research operating system. The OSKit consists of over 1,000,000 lines of code, most of which is being independently maintained by the developers of the "donor" systems. At this time, about 40% of the OSKit has been explicitly converted to Knit units. Although the OSKit is written in C, some parts are distinctly object-oriented: a lightweight subset of Microsoft's COM is used in a number of places. The OSKit has been used to build large systems such as operating system kernels and file servers, to implement advanced languages directly on the hardware, and for smaller projects such as embedded systems and bootloaders.

As an initial example, consider the problem of managing block I/O device drivers, which provide low-level access to block-oriented storage media such as disks and tapes. An operating system is generally configured at build-time to include one device driver for each kind of supported block device: e.g., IDE disk, SCSI disk, and floppy disk drive. At run-time, the operating system queries each driver for information (e.g., the type and capabilities of the driver): the driver discovers the physical devices that it manages, and at the request of the OS, creates run-time objects to represent each of these devices. To make it easy to configure OSKit-based systems with different sets of block device drivers, we apply the *Abstract Factory* pattern as illustrated in Figure 3.3. In OOP terms, we define a common abstract class (BlockDevice) to be supported by all block devices, and we define abstract classes (BlkIO and DriverInfo) for the products that each driver may produce. The actual drivers and products map to concrete classes as shown.

Having identified the pattern at hand, we can now apply the steps of our method to translate the pattern structure into appropriate unit definitions. First (step 1) we identify the abstract classes: clearly, these are BlockDevice, BlkIO, and DriverInfo. Next (step 2): because each device driver can manage multiple physical devices, we need at most one instance of each driver in any system we might build. (We need zero or one, depending on whether or not we choose to support a particular kind of device.) Thus, each of our concrete factories is a static participant. In contrast, since we do not know the number of

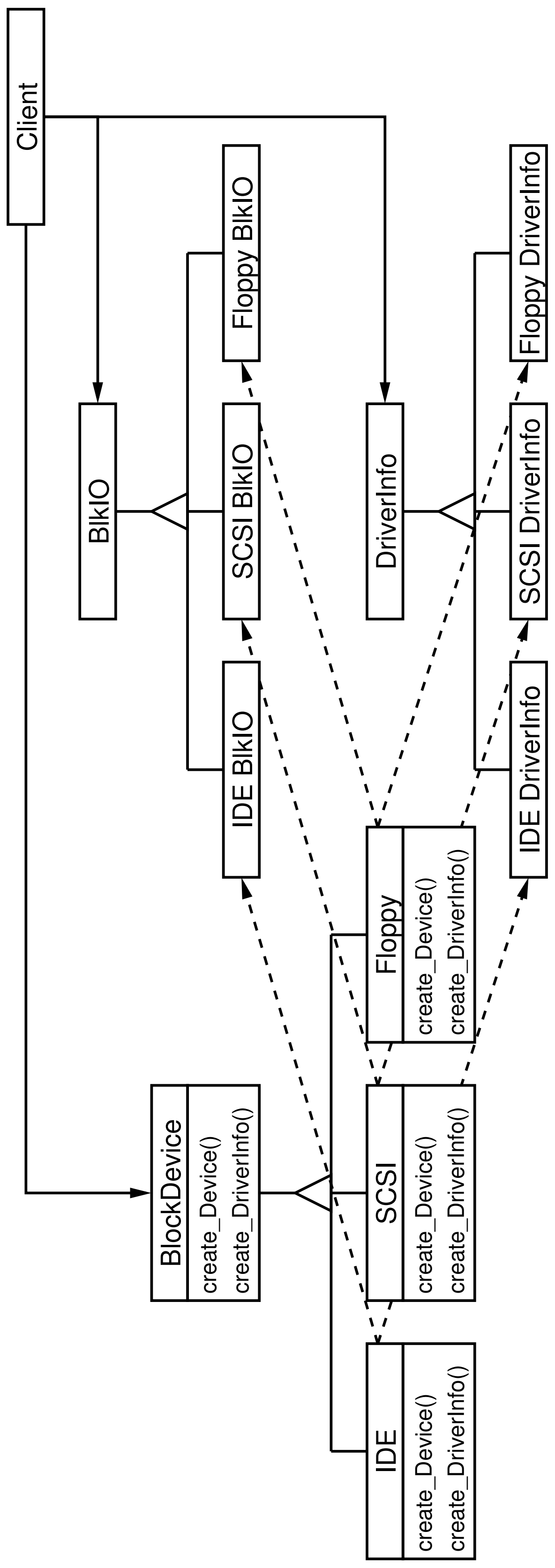


Figure 3.3. Using the *Abstract Factory* pattern to manage block devices in OSKit-based systems

physical devices that will be present at run-time, each concrete product class is a dynamic participant.

We now define the interfaces for our static participants (step 3). The interface to each concrete factory class is defined by the abstract BlockDevice class: we therefore define a corresponding I_BlockDevice interface. As described in Section 3.3.1, we translate the BlockDevice methods into ordinary C functions, because we do not need to represent our static participants as run-time objects.

In defining the interfaces for our dynamic participants (step 4), we need to translate the participant's methods in a way that allows us to identify instances at run-time. Because we are using Knit, we translate the BlkIO and DriverInfo classes into unit port interfaces that include C types for the products. In addition, each product method becomes a C function that takes a run-time instance.

Next (step 5) we create the unit definitions for each of our concrete participants. This is a straightforward mapping from the pattern structure: the exports of each unit are determined by the provided interfaces (i.e., the participants' classes), and the imports are determined by the connections in the pattern structure.

Finally, we create a compound unit in which we instantiate the units that we need, and connect those instances according to the pattern (step 6). For example, to create a system with just IDE support, we would define the unit instances and links shown in Figure 3.4. The unit definitions that we created in steps 1–5 are reusable for many systems, but the structure of the final unit composition in step 6 is often specific to a particular system configuration.

Our method describes the process of creating appropriate unit definitions, but it does not address the problem of unit implementation: i.e., the source code. We have found, however, that appropriate implementation is often straightforward. In the example above, the OSKit units are implemented by existing OS device drivers with little or no modification. Most changes, if needed at all, can be implemented by *Adapter* units that wrap the existing code. Furthermore, the device-specific code can be isolated in the units that define our products. This means that we can write one unit definition for our factory instead of one each for IDE, SCSI, and Floppy. Each instance of this factory imports the units that define a related family of products. Knit's constraint system can be used to statically ensure that the system designer does not accidentally connect a mismatched set of products.

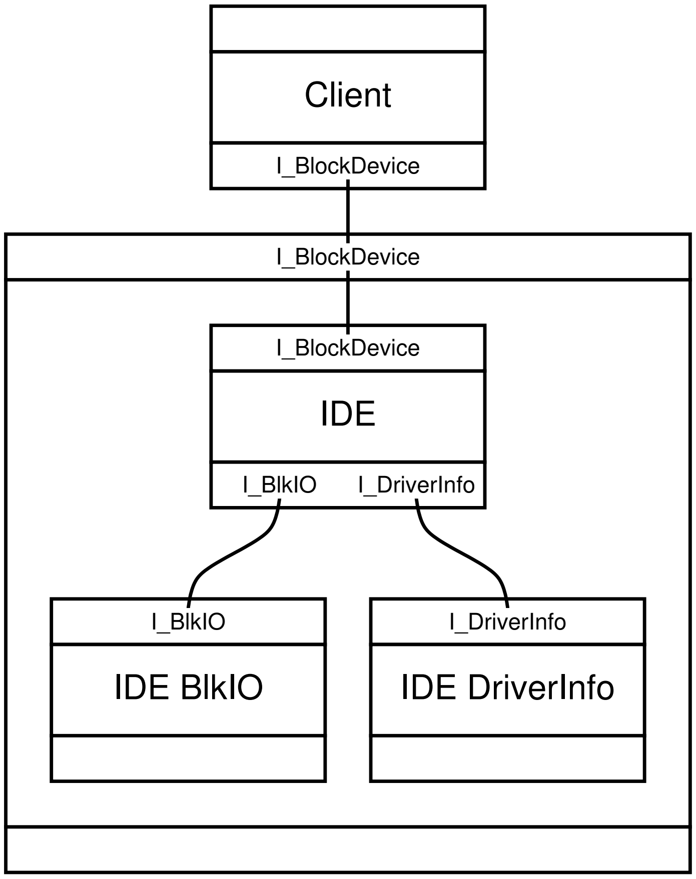


Figure 3.4. Result of applying our method to Figure 3.3

3.3.3 Example: OSKit Filesystems

Having illustrated the method in detail in the previous section, we now show the result of applying the method to a more complex example. Figure 3.5 shows one possible configuration of a filesystem in the OSKit. The primary parts of the system are: Main, an application that reads and writes files; FS Namespace, which implements filepaths (like `/usr/bin/latex`) on top of the more primitive file and directory abstraction; Ext2FS, a filesystem from the Linux kernel distribution; and Linux IDE, a Linux device driver for IDE disks. The other units in the system connect these primary parts according to the *Abstract Factory*, *Adapter*, *Decorator*, *Strategy*, *Command*, and *Singleton* patterns. All participants in these patterns are currently implemented as described with one exception (*Command*) described below.

3.3.3.1 Abstract Factory

Figure 3.5 contains two abstract factories: the Linux IDE and OSEnv/x86 units. (In both cases, only the enclosing compound unit is shown.) The OSKit defines an interface (called the "OS Environment Interface") for all components to use when manipulating interrupts, setting timers, allocating memory, and so on. This interface abstracts the more obtrusive details of the underlying platform. In Figure 3.5, this interface is implemented by OSEnv/x86 for the Intel x86 hardware but we could have chosen OSEnv/StrongARM for the StrongARM architecture or OSEnv/Linux to run as a user-mode Linux program. (The latter choice would necessitate a different choice of device driver.) It is appropriate to fix on a particular platform at this stage because moving to another would require the system to be rebuilt.

3.3.3.2 Adapter

The hybrid nature of the OSKit gives rise to many adapters. The OSEnv$\rightarrow$Linux adapter implements Linux internal interfaces in terms of the OSKit-standard I_OSEnv, allowing us to include Linux-derived units in the system. The LinuxFS$\rightarrow$FS and Linux$\rightarrow$BlkDev adapters implement standard OSKit interfaces for filesystems and block devices using the internal Linux interfaces for these things. Being able to use Linux-derived units is extremely useful for OSKit systems: instead of writing and maintaining new filesystems and device drivers, the OSKit exploits the hard work of the Linux community. The OSKit uses this approach to provide 30 Ethernet drivers, 23 SCSI drivers, and 11 filesystems.

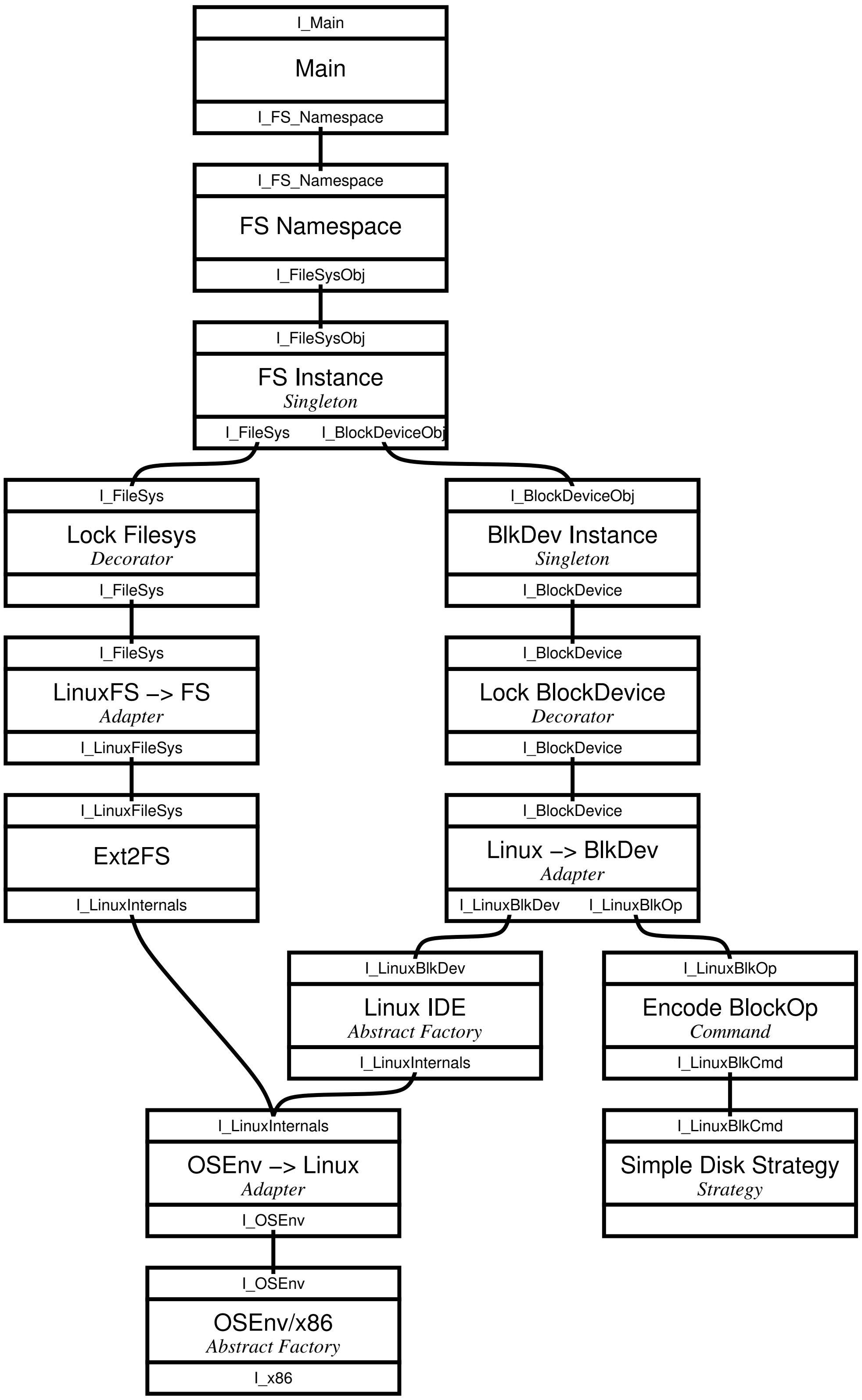


Figure 3.5. A possible configuration of an OSKit filesystem

An interesting part of the LinuxFS→FS and Linux→BlkDev adapters is that they have both static and dynamic aspects. The static aspect adapts the static interfaces of the participants: those used for initialization, finalization, and mounting a filesystem on a disk partition. The dynamic aspect adapts the interfaces of dynamic participants, wrapping Linux block device objects as OSKit block device objects, Linux filesystem objects as OSKit filesystem objects, and Linux file and directory objects as their OSKit equivalents. This illustrates how our approach complements the conventional approach: our units make it apparent which decisions are static (e.g., the decision to use Linux components with OSKit components) and which are dynamic (e.g., how many files will be opened, which files will be opened).

3.3.3.3 Decorator

If this is a multithreaded system, we must take care to acquire and release locks when accessing the filesystem and device driver objects. The decorators Lock Filesys and Lock BlockDevice acquire locks when entering the decorated objects and release locks when leaving.

It would be a serious error to omit one of these lock decorators (leading to race conditions) or to insert it in the wrong place (leading to deadlock), so we use the constraint system to check that they are placed correctly. This may seem like overkill in such a simple configuration, but the reader will appreciate that this is just one of many rules that must be enforced and that we have omitted many units that would appear in a complete system. The complete system—including units for bootstrapping, console I/O, memory allocation, threads and locks, etc.—consists of over 100 unit instances.

3.3.3.4 Strategy

Disk drivers can optimize disk operations by coalescing reads and writes on adjacent blocks and can optimize disk seeks by reordering read and write requests. The series of actual requests issued to the disk is determined by a strategy unit. In Figure 3.5, we have selected the Simple Disk Strategy unit (which queues requests in the order they are issued) but we could have chosen a strategy that coalesces disk operations or reorders requests using an elevator algorithm. (The elevator strategy is not yet implemented.)

3.3.3.5 Command

The Simple Disk Strategy unit manipulates a list of outstanding requests, and these requests are parts of a *Command* pattern. The participants in this pattern are currently integrated within the implementation of the Simple Disk Strategy unit, but could be separated as shown in Figure 3.5 into a separate unit Encode BlockOp which provides a separate function for each kind of request (e.g., read or write). This unit would construct request objects and pass them to Simple Disk Strategy, which would process the requests.

3.3.3.6 Singleton

In this system, we made a design decision to have a single device and a single filesystem instance. One could imagine using a device driver implementation that supports just one instance of that device type or a filesystem implementation that supports just one instance of that filesystem type. But this is not what Linux components do. Most Linux device drivers and filesystems are written to support multiple instances of a device or filesystem. To overcome this mismatch, we use the BlkDev Instance and FS Instance units that each create and manage a single instance of the corresponding dynamic objects. These units are effectively adapters, making dynamic pattern participants appear as if they were static. This mismatch is common in reuse and maintenance scenarios: the cost of making changes influences the choice of design. Our approach to patterns addresses such real-world concerns.

3.3.4 Discussion

The previous sections demonstrate our approach to utilizing design patterns in the context of two example systems. In both examples we had a mix of static and dynamic participants: the static participants were realized by unit instances corresponding to "object instances" while the remaining dynamic participants were realized by units that create the pattern participant objects at run-time. In both examples we were able to lift a great deal of static knowledge to the unit level, but the exact amount depended on the patterns and their application to the particular design problems at hand.

In general, the static and dynamic parts of many patterns will vary from situation to situation. However, *in common use*, most pattern structures contain many participants and connections that are in fact static: these parts can be fruitfully lifted out of the participants' source implementations and then managed at the level of units. To test this claim, we analyzed the structures of all of the patterns described in Gamma et al.'s *Design Patterns* catalog [13]. For each, we considered common uses of the pattern in component-based systems software such as that built with the OSKit. We then applied our method to translate the pattern structures into appropriate units and unit parts.

Table 3.1 summarizes the results of our analysis. Each row of the table shows the translation of the participants within a single pattern. Overall, the table shows that most participants frequently correspond to static, design-time information and are therefore realizable within our unit model as design-time entities. (These are the columns under the "Design-Time/Static Participants" heading.) Abstract classes map naturally to unit interfaces. Participants that are singletons within the context of a pattern map naturally to unit instances that implement these participants. In some cases, a participant both defines an interface and represents a concrete instance, as indicated in the table. For example, in the *Facade* pattern, the Facade entity has both interface and implementation roles. In some cases, the designer may choose to implement a particular participant in more than one way: for instance, the designer may choose to implement a Client participant as a unit instance, or as a set of ports that allow the client to be connected at a later stage of the overall design. In other cases, the appropriate implementation of one participant depends on the characteristics of another: in the *Decorator* pattern, for example, the appropriate realizations of Decorator and Concrete Decorator differ according to the "singleton-ness" of the Concrete Component. Where the common translation or use varies, we have indicated this with italics, and we list the participant in multiple categories.

Because the OSKit is such a large body of code, largely derived from systems not explicitly organized around patterns, it is difficult to identify all uses of a particular pattern and so it is hard to determine the ratio of static uses to dynamic uses. With that caveat, we have found static instances of all three categories of patterns (creational, structural, and behavioral) in OSKit-based systems, but most examples are either creational or structural. Our admittedly incomplete study failed to find static examples of some patterns including *Flyweight*, *Chain of Responsibility*, and *Visitor*.

In summary, Table 3.1 shows that our approach to realizing patterns is applicable to many patterns: most have common applications in which many or all of the participants represent static system design knowledge that can be utilized by tools for design rule checking, code generation, and system optimization. This applies even when a participant is dynamic and is realized by a unit that produces objects at run-time. In these cases, we can use the unit model to define our run-time classes/types in terms of mixins, thus increasing the potential reuse of our unit definitions and implementations.

Table 3.1. Summary of how the participants within the *Design Patterns* catalog [13] can be realized within the unit model, for common situations in the design of component-based, C language systems software. Participants are classified according to their common and primary realizations; certain uses of patterns will dictate different realizations. Where common use varies, participants are italicized and are listed in all applicable categories. Some participants have both interface and implementation roles as shown. Participants that map to unit instances usually also require interface definitions to describe their ports.

	Design-Time/Static Participants			**Dynamic Participants**
Pattern	**Realized By Unit Interface (Method Steps 1, 3, 4)**	**Realized By Unit(s) Impl'ing the Instance(s) (Method Steps 2, 5)**	**Realized By Port(s) On Unit Instances (Method Step 5)**	**Realized By Unit Defining the Class (Method Steps 2, 5)**
Abstract Factory	Abstract Factory Abstract Product	Concrete Factory *Client*	*Client*	Concrete Product
Builder	Builder	Concrete Builder Director		Product
Factory Method	Product Creator	Concrete Creator		Concrete Product
Prototype	Prototype	*Client*	*Client*	Concrete Prototype
Singleton		Singleton		
Adapter (class)	Target	*Client*	*Client*	Adaptee Adapter
Adapter (object)	Target	*Client* Adaptee Adapter	*Client*	
Bridge	Abstraction (intfc.) Refined Abstraction (intfc.) Implementor	Abstraction (impl.) Refined Abstraction (impl.) Concrete Implementor		
Composite	Component	*Client*	*Client*	Leaf Composite
Decorator	Component	*Concrete Component* *Concrete Decorator*	*Decorator*	*Concrete Component* *Decorator* *Concrete Decorator*

Table 3.1. Continued

Pattern	Design-Time/Static Participants			Dynamic Participants
	Realized By Unit Interface (Method Steps 1, 3, 4)	**Realized By Unit(s) Impl'ing the Instance(s) (Method Steps 2, 5)**	**Realized By Port(s) On Unit Instances (Method Step 5)**	**Realized By Unit Defining the Class (Method Steps 2, 5)**
Facade	Facade (intfc.)	Facade (impl.) subsystem classes		
Flyweight	Flyweight	Flyweight Factory *Client*	*Client*	Concrete Flyweight Unshared Conc. Flyweight
Proxy	Subject	Proxy Real Subject		
Chain of Resp.	Handler	Concrete Handler *Client*	*Client*	
Command	Command	*Client* Invoker Receiver	*Client*	Concrete Command
Interpreter	Abstract Expression	Context *Client*	*Client*	Terminal Expression Nonterminal Expression
Iterator	Iterator Aggregate	*Concrete Aggregate*		Concrete Iterator *Concrete Aggregate*
Mediator	Mediator	Concrete Mediator colleague classes		
Memento		Originator *Caretaker*	*Caretaker*	Memento
Observer	Subject (intfc.) Observer	Subject (impl.) Concrete Subject *Concrete Observer*	*Concrete Observer*	
State	Context (intfc.) State	Context (impl.) Concrete State		
Strategy	Strategy	Concrete Strategy		

Table 3.1. Continued

	Design-Time/Static Participants			Dynamic Participants
Pattern	**Realized By Unit Interface (Method Steps 1, 3, 4)**	**Realized By Unit(s) Impl'ing the Instance(s) (Method Steps 2, 5)**	**Realized By Port(s) On Unit Instances (Method Step 5)**	**Realized By Unit Defining the Class (Method Steps 2, 5)**
		Context		
Template Method		Abstract Class Concrete Class		
Visitor	Visitor Element	*Concrete Visitor* *Object Structure*		*Concrete Visitor* Concrete Element *Object Structure*

3.4 Analysis

The key feature of our approach is that we express static pattern relationships in a component *configuration* language instead of expressing those relationships in the component *implementation* language. In this section, we detail the benefits and costs of this separation of concerns.

3.4.1 Benefits of Our Approach

Our technique for realizing patterns has three main consequences. First, because static pattern information is located in single place (our compound units) and because component interconnections are fully resolved at build-time, it is possible for tools to perform a more thorough analysis of the software architecture than in the conventional approach to realizing patterns. Second, because the unit language has a single purpose—to express components, their instantiations, and their interconnections—it is possible to provide features in the language that make this task easier. Third, because the task of pattern composition is moved out of the implementations of the participants, those implementations can be simpler and are less likely to be hard-wired to function only in fixed pattern roles. These three consequences lead to benefits in the areas of error detection, performance, and ease of understanding and reuse, which we explore in the following sections.

3.4.1.1 Checking Architectural Constraints

In the conventional approach to realizing design patterns, it can be difficult to enforce static system design constraints: the rules are encoded "implicitly" in the implementation, making them difficult for people to find and for tools to enforce in the face of future system evolution. Our approach to realizing patterns has the following advantages over the conventional method.

The constraint checker detects global, high-level errors. The constraint checker within the Knit unit compiler can detect "global" errors that involve many parts of a system, whereas a conventional language type system is restricted to detecting relatively local errors. Such global constraints often deal with high-level system composition issues—e.g., ensuring that domain-specific properties hold across many interconnected components—whereas conventional type systems and tools are restricted to detecting relatively low-level and general types of errors such as uncaught exceptions [1], dereferenced null pointers [7], and race conditions [9].

Constraints express domain-specific design rules. A software architect is often interested in detecting domain-specific errors. For example, recent versions of RTLinux [24] permit normal (user-level) application code to be called from a hard real-time kernel. Without going into detail, an essential requirement of such applications is that they never invoke a system call while running in real-time mode. We have used Knit's constraint system to check this constraint for RTLinux applications: i.e., to verify, at compile-time, that there are no paths from an application's real-time signal handler into the Linux kernel.

Design errors are separated from implementation errors. In particular, this reduces the level of expertise required in order to use (or reuse) a component correctly, inside or outside of a pattern.

The constraint checker need not deal with the base implementation language. Our constraint checker deals only with the unit specification language, not with the source code of the components. Because the unit language is simple, the constraint checker is simple and precise. Further, it would be easy to extend with more powerful and perhaps more pattern-specific reasoning methods in the future. In contrast, to detect design errors in a conventionally realized design pattern, a tool would need to deal with all the complexities of the base implementation language: loops, recursion, exceptions, typecasts, virtual functions, pointers, and so on. Such a tool is therefore difficult to create—greatly raising the barrier to developing domain-specific analyses—and is often imprecise.

Many architecture description languages can provide the advantages described above: like our tools, they achieve this by separating the description of the architecture from the implementation of the components, and by being domain-specific instead of general-purpose. Bringing these features to bear on the realization of design patterns is one of the strengths of our tools and approach.

3.4.1.2 Performance Optimization

Another strength of our approach is that static pattern knowledge is readily available for system optimization. The conventional approach to realizing patterns uses language features that introduce indirections to achieve greater flexibility. These indirections—principally indirect function calls—impose a performance penalty that can often be avoided in our approach.

Static implementation enables many optimizations. When component instances are connected statically, indirect function calls are often turned into direct calls. This affords the compiler the opportunity to inline function calls, thus eliminating overhead and exposing additional and often more significant opportunities for optimization, especially those that specialize a function for a particular context. In addition, highly optimizing compilers, or compilers aided by a source transformation that Knit can perform, are able to inline functions across module boundaries. In previous work [21], we used Knit to implement a network router made of very small components. (Each packet traversed 10–20 components while being forwarded.) Applying cross-component inlining eliminated the cost of many function calls but, more significantly, enabled the C compiler to apply all of its intra-procedural optimizations. The overall effect of this optimization was to reduce the execution time of the routing components by 35%.

Static implementation makes performance less sensitive to code changes. To eliminate virtual function calls, a compiler requires a global (or near global) analysis of the program being optimized. These analyses are necessarily affected by subtle features of how the program is expressed: a consequence is that any change to that program could potentially change the analysis result and therefore change whether or not the optimization can be applied. In a performance-sensitive situation (e.g., in real-time code), loss of an optimization may affect program correctness. By making static knowledge explicit, our approach to patterns helps to reduce the complexity of the resulting system, thus promoting compile-time analysis and making "global" performance less sensitive to local code changes.

3.4.1.3 Ease of Understanding and Code Reuse

In the conventional approach to realizing design patterns, one takes into account the role of each participant when implementing the participant—or, if the pattern is applied after implementation, one modifies the participant to reflect their roles in the pattern. In our approach, because units do not contain direct references to other participants, units often need no modification in order to be used in a particular role in a pattern. Avoiding even small changes to the participants leads to significant benefits.

The approach is usable when code cannot be changed. The implementation of a participant may be unchangeable if the code has multiple users with different needs, if the source code is not available, or if the code is being actively maintained by a separate organization. For instance, the developers of the OSKit cannot practically afford to change the Linux components that they incorporate: they must deal with the code as it is written.

A participant can be used in multiple patterns. Separating a participant's role from its implementation is beneficial when the implementation can be "reused" to serve in many different roles, perhaps concurrently in several different patterns. The unit model allows a programmer to separate a participant's primary implementation from any code needed to adapt that implementation to a particular pattern role: by creating a unit composition, a programmer can "weave" code at the imports and exports of a participant unit instance.

Code is not obfuscated with indirections. The conventional realization of a design pattern often achieves flexibility by introducing additional levels of indirection that are apparent in the implementations of the participants. This indirection can obscure the primary purpose of the code. For example, before applying the unit model to the OSKit, we relied on objects, factories, and registries to enable reconfiguration. Over time, much OSKit code came to look like the following:

```
clientos = registry->lookup(registry, clios_iid);
fsn      = clientos->create_fsnamespace(filesys);
file     = fsn->lookup_path("/usr/bin/latex");
```

The code was often further complicated to deal with run-time errors. In any particular system, however, the values of `clientos` and `fsn` were fixed in the system configuration, and knowable at compile-time. After applying our approach, such code could often be simplified to just:

```
file = lookup_path("/usr/bin/latex");
```

making it clear that the selection of `lookup_path`'s implementation is a static, not dynamic, system property.

3.4.2 Costs of Our Approach

Our approach to realizing design patterns is not appropriate for all situations and design problems. The following paragraphs summarize the costs and potential problems of our approach.

Our approach only specifies the static parts of patterns. The main goal of our approach is to use an external component language to specify the static aspects of system architecture. It is inappropriate (and often infeasible) to use our approach to specify fundamentally dynamic elements of software architecture.

Our approach commits code to being static or dynamic. One can imagine that having carefully used our approach (with its emphasis on static participants and relationships) to realize a pattern, a change of requirements might turn a relationship from static to dynamic, requiring that the pattern be re-implemented using the conventional object-oriented approach (with its emphasis on dynamic participants and relationships). This is a problem: while it is easy to use a dynamic system in a static situation, it is not so easy to use a static system in a dynamic way. Therefore, when using our approach, one should design systems in such a way that expected changes in the system requirements are unlikely to require changing the static and dynamic natures of pattern participants—but we recognize that this is not always possible. An automated implementation of our approach (perhaps based on partial evaluation [6]) could partly solve this problem by transforming dynamic code into static code, although this would not promote the benefit of easier writing and understanding of code through eliminated indirections.

Our approach requires support for the unit component model. To fully benefit from our approach, one needs language support in the form of an advanced module or component system and, ideally, a constraint checking system. This implies several costs: one must switch to using new tools, learn the component definition and linking language, learn to use the constraint checking language, and convert existing codebases to use the component language. This can be a significant undertaking. As described in Section 3.2, however, it is possible to use existing tools and techniques to achieve some (but not all) of the benefits of the unit component model.

Our approach can obscure the differences between patterns. When one looks at the unit diagrams of participants and relationships, it is clear that sometimes, different patterns look the same when realized in our approach. However, this observation is also true of the conventional approach to patterns: many patterns are realized in similar ways but differ significantly in their purpose.

3.5 Related Work

Gamma et al.'s *Design Patterns* book [13] triggered a flurry of papers on implementing patterns in object-oriented languages. Here, we consider representatives of particular styles of implementation. Bosch [3] describes a language LayOM for constructing C++ classes by adding a number of layers to a simple class. By using layers corresponding to particular patterns, Bosch solves the *traceability* problem—that it is hard to find and identify patterns in one's code—and enables pattern implementations to be reused. However, because the layers form part of the class description, the role of each pattern participant is hardwired and the participants cannot be used in other patterns without being modified. Bosch makes no mention of static analysis, detecting design errors, or optimization. Marcos et al. [16] describe an approach that is closer to ours: the code that implements participants is clearly separated from the code that defines their roles in patterns. The difference is that their approach is based on run-time reflection within a metaprogramming system (CLOS), and so they do not support static analysis or optimization. Tatsubori and Chiba [23] describe a similar approach to that of Marcos et al., except that it uses OpenJava's compile-time metaprogramming features. Like Marcos et al., they separate roles from participants and, because they use compile-time metaprogramming, it should be possible to perform static analysis. However, OpenJava does not provide anything like Knit's unit constraint system.

Krishnamurthi et al. [15] describe an approach to pattern implementation based on McMicMac, an advanced macro system for Scheme. Their approach is like that of Tatsubori and Chiba: patterns are expanded statically (enabling optimization) and the application of patterns is not separated from the definitions of the participants. Unlike OpenJava, McMicMac provides source-correlation and expansion-tracking facilities that allow errors to be reported in terms of the code that users wrote instead of its expansion, but there is no overall framework for detecting global design errors.

Baumgartner et al. [2] discuss the influence of language features on the implementation of design patterns. Like us, they note that Gamma et al.'s pattern descriptions [13] would be very different in a language that directly supports abstract interfaces and a module mechanism separate from class hierarchies. Baumgartner also lists a number of other useful features including mixins and multimethods. MultiJava [5] adds some of these features to Java, enabling them to cleanly support the *Visitor* pattern and to describe "open classes." Our colleagues' paper on Jiazzi [17] shows how the open class pattern can be realized with units. Bruce et al. [4] describe virtual types and show how they apply to the *Observer* pattern. All of these papers describe language features that address problems in implementing patterns in object-oriented languages, but their focus is on the technology, not the approach enabled by that technology.

At the other end of the spectrum, there are component programming models, module interconnection languages (MILs) [20], and architecture description languages (ADLs) [18]. Our implementations of the unit model lie at the intersection of these three approaches. Units are like COM or CORBA components except that units play a more static role in software design; units are like MILs in that each implementation of the unit model supports just one kind of unit interconnection; and units are like ADLs in that units support static reasoning about system design.

Module interconnection languages are perhaps closest in purpose to the unit model. The best example we know of using a MIL in the way this paper suggests is FoxNet [14], a network stack that exploits ML's powerful module language. However, although FoxNet clearly uses a number of patterns, there is no explicit statement of this fact and consequently no discussion of implementing a broad range of patterns using a MIL.

Architecture description languages provide a similar but higher-level view of the system architecture to MILs. This higher-level view is the key difference. ADLs describe software designs in terms of architectural features, which may include patterns. ADLs may also provide implementations of these features: the details of implementation need not concern the user. In contrast, this paper is all about those implementation issues: we describe a method that ADL implementors could apply when adding new patterns to the set provided by their ADL. That said, ADLs provide more expressive languages for describing design rules, specifying components, and reasoning about system design than is currently supported by the unit model. We plan to incorporate more high-level ADL features into our unit languages in the future.

3.6 Conclusion

Design patterns can be realized in many ways: although they are often described in object-oriented terms, a pattern need not always be realized in an OOP language nor always with objects and interconnections created at run-time. In this paper we have presented a complementary realization of design patterns, in which patterns are statically specified in terms of the unit model of components. While this approach is not applicable to all software architectures, it can yield benefits when applied to static systems, and to static aspects of dynamic systems. These benefits include verification of architectural constraints on component compositions, and increased opportunities for optimization between components.

3.7 Acknowledgments

We thank Mike Hibler for his expert help with the OSKit, and Sean McDirmid for many fruitful discussions about the unit model and design patterns. Matthew Flatt, Jason Baker, David Coppit, and Joe Loyall provided valuable comments on drafts of this paper.

We also thank the anonymous reviewers for their many helpful suggestions.

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

3.8 References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, second edition, 1998.

[2] G. Baumgartner, K. Läufer, and V. F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD–TR–96–020, Department of Computer Sciences, Purdue University, 1996.

[3] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[4] K. B. Bruce and J. C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Electronic Notes in Theoretical Computer Science*, volume 20. Elsevier Science Publishers, 2000.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. of the 2000 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 130–145, Minneapolis, MN, Oct. 2000.

[6] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. Partial evaluation for software engineering. *ACM Computing Surveys*, 30(3es), Sept. 1998.

[7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 1–16, San Diego, CA, Oct. 2000.

[8] 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.

[9] C. Flanagan and S. N. Fruend. Type-based race detection for Java. In *Proc. of the ACM SIGPLAN '00 Conf. on Programming Language Design and Implementation (PLDI)*, Vancouver, Canada, June 2000.

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

[11] M. Flatt and M. Felleisen. Units: Cool units for HOT languages. In *Proc. of the ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation (PLDI)*, pages 236–248, Montreal, Canada, June 1998.

[12] 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.

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

[14] 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.

[15] S. Krishnamurthi, Y.-D. Erlich, and M. Felleisen. Expressing structural properties as language constructs. In *Programming Languages and Systems (Proc. of the Eighth European Symp. on Programming, ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 258–272. Springer-Verlag, Mar. 1999.

[16] C. Marcos, M. Campo, and A. Pirotte. Reifying design patterns as metalevel constructs. *Electronic Journal of SADIO*, 2(1):17–29, 1999.

[17] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of the 2001 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, pages 211–222, Tampa, FL, Oct. 2001.

[18] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Jan. 2000.

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

[20] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4), Nov. 1986.

[21] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, Oct. 2000.

[22] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[23] M. Tatsubori and S. Chiba. Programming support of design patterns with compile-time reflection. In *Proc. of the OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, pages 56–60, Vancouver, Canada, Oct. 1998.

[24] V. Yodaiken. The RTLinux manifesto. In *Proc. of the Fifth Linux Expo*, Raleigh, NC, Mar. 1999.

PART III

DYNAMIC CPU MANAGEMENT FOR REAL-TIME, MIDDLEWARE-BASED SYSTEMS

CHAPTER 4

DYNAMIC CPU MANAGEMENT FOR REAL-TIME, MIDDLEWARE-BASED SYSTEMS

Many real-world distributed, real-time, embedded (DRE) systems, such as multi-agent military applications, are built using commercially available operating systems, middleware, and collections of pre-existing software. The complexity of these systems makes it difficult to ensure that they maintain high quality of service (QoS). At design time, the challenge is to introduce coordinated QoS controls into multiple software elements in a noninvasive manner. At run time, the system must adapt dynamically to maintain high QoS in the face of both expected events, such as application mode changes, and unexpected events, such as resource demands from other applications.

In this paper we describe the design and implementation of a *CPU Broker* for these types of DRE systems. The CPU Broker mediates between multiple real-time tasks and the facilities of a real-time operating system: using feedback and other inputs, it adjusts allocations over time to ensure that high application-level QoS is maintained. The broker connects to its monitored tasks in a noninvasive manner, is based on and integrated with industry-standard middleware, and implements an open architecture for new CPU management policies. Moreover, these features allow the broker to be easily combined with other QoS mechanisms and policies, as part of an overall end-to-end QoS management system. We describe our experience in applying the CPU Broker to a simulated DRE military system. Our results show that the broker connects to the system transparently and allows it to function in the face of run-time CPU resource contention.

4.1 Introduction

To meet the requirements of the market, real-time and embedded software systems must increasingly be designed atop commercial, off-the-shelf (COTS) operating systems and middleware. These technologies promote rapid software development by allowing system developers to concentrate on their application logic rather than on low-level "infrastructural" code. In addition, commercial operating systems and middleware promote software quality by providing tested, efficient, and reliable implementations of low-level functionality. Finally, these technologies promote scalability across different types of embedded platforms, configurability of features and feature selection, and evolvability of the embedded software systems over time. These so-called "-ilities" are essential in a world where embedded system technologies change rapidly and where the high cost of software development must be amortized over several products, i.e., across the life cycle of a product family rather than the lifetime of a single product.

COTS operating systems and middleware are also increasingly required to support the development of distributed, real-time, embedded (DRE) systems. Many real-time systems are built containing multiple processors or processing agents, either tightly connected (e.g., within an automobile or aircraft) or loosely connected (e.g., multiplayer networked games, sensor networks, and networked military systems). Middleware such as CORBA [1] promotes the development of these systems by providing high-level and scalable abstractions for communication between multiple processes. Real-time middleware, such as RT CORBA [2], also provides high-level and portable abstractions for scheduling resources for real-time tasks.

Even with modern middleware, however, it can be a significant software engineering challenge for system developers to design and build DRE systems that meet their real-time requirements. First, because the parts of an embedded software system must often be designed to be reusable across many products, the code that implements real-time behavior for any particular system must be decoupled from the "application logic" of the system's parts. Decoupling makes it possible to collect the real-time specifications for all of the system's parts in a single place—in other words, to modularize the real-time behavior of the system—but leads to the new problem of reintroducing that behavior into the software. Second, even if the implementation of real-time behavior is modularized, developers are challenged with specifying the desired behavior at all. It is a common problem for the execution times of parts of a system to be data-dependent, mode-dependent, configuration-dependent, unpredictable, or unknown. In a distributed real-time system, the sets of communicating tasks and available processor resources may not be known until run time, or may change as the system is running. In sum, the challenges of implementing real-time behavior in many systems include not only decoupling and modularizing of the behavior, but the ability to describe a variety of policies in a high-level and tractable manner, and ensuring that the system continues to operate (perhaps at reduced capacity) in the face of events that occur at run time, both expected and unexpected.

To address these challenges, we have designed and implemented a novel *CPU Broker* for managing processor resources within real-time systems. Our CPU Broker is a CORBA-based server that mediates between the multiple real-time tasks and the facilities of a real-time operating system, such as TimeSys Linux [3]. The broker addresses design-time challenges by connecting to its managed tasks in a noninvasive fashion and by providing an expressive and open architecture for specifying CPU scheduling policies. The broker can manage resources for both CORBA and non-CORBA applications. At run time, the broker uses feedback and other inputs to monitor resource usage, adjust allocations, and deal with contention according to a configured policy or set of policies. The broker is configured at run time through a command-line tool or via invocations on the CORBA objects within the broker: policies are easily set up and changed dynamically. Finally, the broker is designed to fit into larger, end-to-end architectures for quality of service (QoS) management. A single instance of the broker manages CPU resources on a single host, but because its architecture is open and extensible, the broker's policies can be directed by higher-level QoS systems like QuO [4]. This enables coordination of brokers on multiple hosts, coordination with other resource managers, and cooperation with application-level QoS such as dynamic adaptation strategies.

Our CPU Broker was designed to ensure that the CPU demands of "important" applications are satisfied insofar as possible, especially in the face of dynamic changes in resource requirements and availability, in the set of managed tasks, and in the relative importances of the tasks. We have evaluated the broker in these situations through microbenchmarks and synthetic application scenarios. In addition, we have applied and evaluated the CPU Broker in the context of a simulated DRE military application. Our results show that the broker correctly allocates CPU resources in our synthetic tests and in a "real world" system of target-seeking unmanned aerial vehicles (UAVs). The broker connects to applications in a transparent fashion and improves the ability of the UAV system to identify targets in a timely manner.

The primary contributions of this paper are threefold. First, we describe our architecture for dynamic CPU management in real-time systems: an architecture that addresses the critical software engineering challenges of specifying and controlling real-time behavior in the presence of anticipated and unanticipated events. Second, we present our CPU Broker, which effectively implements our architecture atop a commercial RTOS and industry-standard middleware, enables noninvasive integration, and provides an open and extensible platform for CPU management. Finally, we demonstrate the use of our CPU Broker and evaluate its performance in both synthetic scenarios and a simulated DRE military application. Our results show that the broker approach can effectively address both the design-time and run-time challenges of managing real-time behavior in COTS-based real-time systems.

4.2 Related Work

A great deal of work has been done in the areas of feedback-driven scheduling, real-time middleware, and middleware-based QoS architectures. In this section we summarize representative work in each of these areas and compare it to the CPU Broker. In general, our focus has been to improve on previous work by addressing the practical and software engineering barriers to deploying adaptive, feedback-driven scheduling in modern COTS-based embedded and real-time systems. These goals are elaborated in our previous work [5].

In the area of feedback-driven scheduling, Abeni and Buttazzo [6] describe a *QoS Manager* that is similar to our CPU Broker. The QoS Manager handles requests from three types of real-time tasks: pseudo-proportional-share tasks, (periodic) multimedia tasks, and (aperiodic) event-driven tasks. Tasks are weighted with importance values, and the QoS Manager uses feedback from multimedia and event-driven tasks to adjust those tasks' scheduling parameters. Our work differs from theirs in three important ways. First, whereas the QoS Manager is implemented within the HARTIK research kernel, our CPU Broker is built atop a COTS operating system and industry-standard middleware. (In later work [7], Abeni et al. implemented and analyzed a feedback-driven scheduler for Linux/RK [8].) Second, our focus is on noninvasive approaches: Abeni and Buttazzo do not describe how to cleanly separate the feedback from the applications being controlled. Third, our CPU Broker is based on an open architecture, making it easy to implement new policies, inspect the broker, or otherwise extend it. The QoS Manager, on the other hand, has a more traditional, monolithic architecture. Similar differences distinguish our CPU Broker from Nakajima's adaptive QoS mapping system [9], which was used to control video streaming applications under Real-Time Mach.

In the area of real-time middleware, there has been significant work in both commercial standards and novel research. For instance, the Object Management Group has defined and continues to evolve standards for RT CORBA [2, 10]. These standards are concerned with issues such as preserving thread priorities between clients and servers in a distributed system and proper scheduling of I/O activities within an ORB. This is in contrast to our CPU Broker, which manages the resources of a single host, is feedback-driven, and which operates above the ORB rather than within it. RT CORBA is similar to our CPU Broker, however, in that both provide an essential level of abstraction above the real-time services of an underlying operating system: it would be interesting to see if the (priority-based) RT CORBA and the (reservation-based) CPU Broker abstractions could be used in combination in the future. Other researchers have incorporated feedback-driven scheduling into real-time middleware: for example, Lu et al. [11] integrated a *Feedback Control real-time Scheduling service* into nORB, an implementation of CORBA for networked embedded systems. Their service operated by adjusting the rate of remote method invocations on server objects: i.e., by adjusting the clients to match the resources of the server. Our CPU Broker, on the other hand, would adjust the resources available to the server in order to meet its clients' demands. Our approaches are complementary: a robust DRE system might use both client-side and server-side adaptation effectively, and our CPU Broker is open to integration with other QoS adaptation mechanisms. Finally, it should be noted that our CPU Broker can manage both middleware-based and non-middleware-based processes, in contrast to the systems described above.

While many middleware-based QoS architectures operate by mediating between applications and an operating system, other architectures are based on applications that can adapt themselves to changing resource availability. The *Dynamic QoS Resource Manager* (DQM) by Brandt et al. [12] is of this second type and is perhaps the most similar to our CPU Broker in both its goals and approach. Like our broker, DQM is implemented as a middleware server atop a commercial OS. It monitors a set of applications, and based on CPU consumption and availability, it tells those tasks to adjust their requirements. The primary difference with our CPU Broker is in the level of adaptation. Our broker changes (and enforces) tasks' CPU allocations by interfacing with an RTOS. DQM, on the other hand, requests (but cannot enforce) that applications switch to new "execution levels," i.e., operational modes with differing resource needs. DQM and the CPU Broker implement complementary approaches to CPU management, and it would be interesting to combine our broker with a framework for benefit-based, application-level adaptation like DQM. Several researchers, including Abeni and Buttazzo [13], have demonstrated the benefits of combining adaptive reservation-based scheduling with application-specific QoS strategies.

The CPU Broker uses QuO [4] to connect to CORBA-based applications in a noninvasive manner and to integrate with other QoS management services such as application-level adaptation. QuO provides an excellent basis for coordinating multiple QoS management strategies in middleware-based systems, both for different levels of adaptation and for different resource dimensions. For example, Karr et al. [14] describe how QuO can coordinate application-level and system-level adaptations, e.g., by dropping video frames and reserving network bandwidth. More recently, Schantz et al. [15] demonstrated that the distributed UAV simulation (described in Section 4.5.3) could be made resilient to both communication and processor loads by applying network reservations in combination with our CPU Broker.

4.3 Design

The conceptual architecture of the CPU Broker is illustrated in Figure 4.1, which depicts a sample configuration of the broker. At the top of the figure are the set of tasks (e.g., processes) being managed by the broker. Under those, within the dashed box, are the objects that make up the CPU Broker. As described later in Section 4.4, these objects are normally located all within a single process, but they do not have to be. The broker mainly consists of two types of objects, called *advocates* and *policies*, that implement per-application adaptation and global adaptation, respectively.

4.3.1 Advocates

As shown in Figure 4.1, every task under the control of the CPU Broker is associated with one or more *advocate* objects. The purpose of an advocate is to request CPU resources on behalf of a task. More generally, an advocate transforms an incoming request for resources into an outgoing request.

The primary input to an advocate is a request for a periodic CPU reservation: i.e., a period and an amount of CPU time to be reserved in each period. This is shown as the arrow entering the top of each advocate. On a regular basis—usually, at the end of every task cycle—the topmost advocate for a task receives a report of how much CPU time was consumed by the task since its last report: this is called the *status*. (The details of obtaining a task's status are described in Section 4.4.) The status amount may be more than what

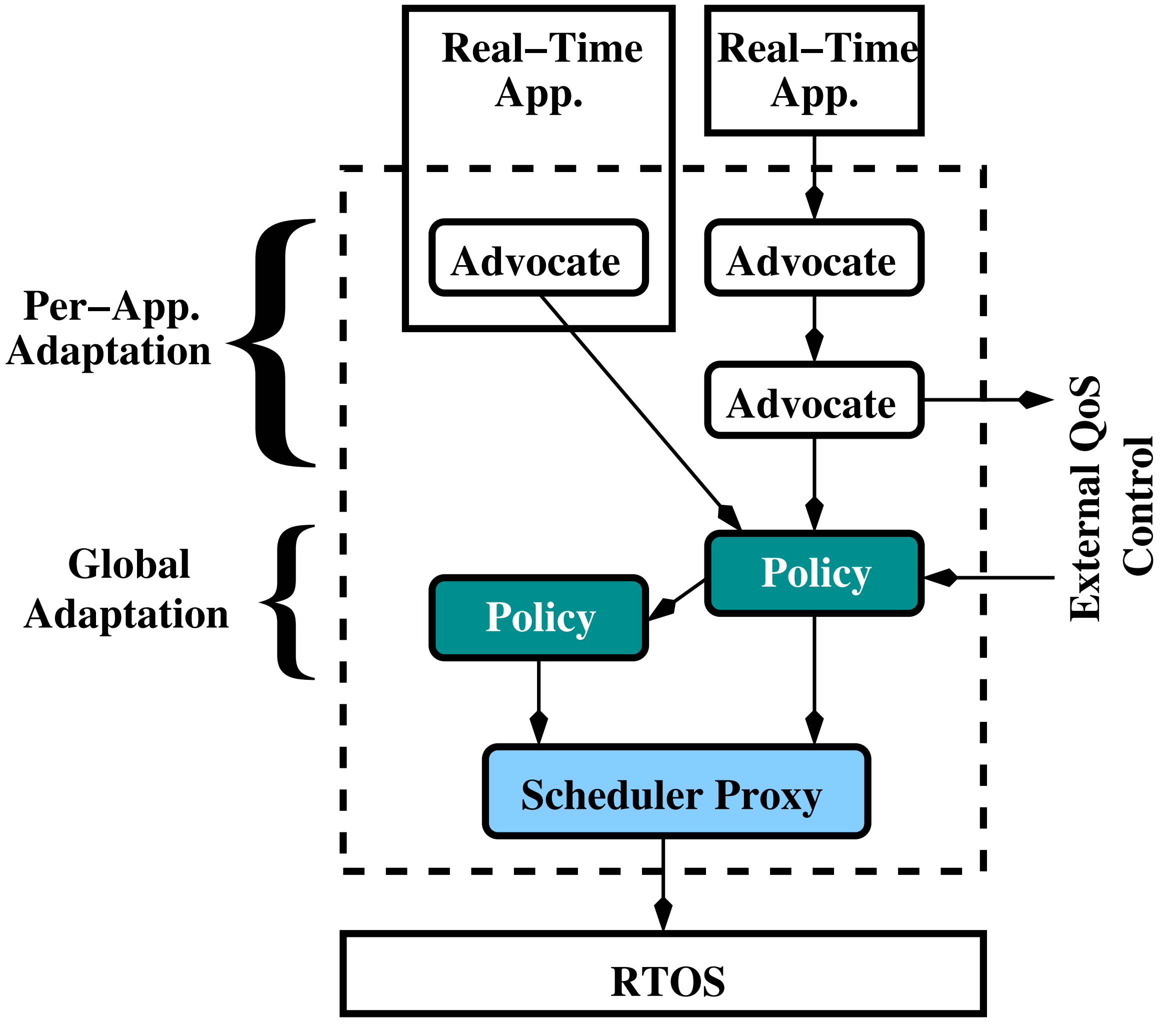


Figure 4.1. Overview of the CPU Broker architecture

is currently reserved for the task: the CPU Broker normally manages "soft" reservations, which allow tasks to consume unreserved CPU resources in addition to their own reserves.

From this information, the topmost advocate decides how its task's reservation should be adjusted. The CPU Broker provides a family of different advocates, implementing different adaptation algorithms, and is an open framework for programmers who need to implement their own. A typical advocate works by producing a reservation request that closely matches its task's observed behavior. For instance, if the status amount is greater than the task's currently reserved compute time, the advocate would request a new reservation with an increased compute time that (better) meets the task's demand. If the status amount is smaller, on the other hand, then the requested compute time would be reduced. The details of the adaptation strategies are up to the individual advocates; for instance, different advocates may use historical data or may adapt reservations quickly or slowly. Advocates can also take input from a variety of sources in order to implement their policies, as illustrated in Figure 4.2. For instance, our architecture allows an embedded systems designer to deploy an advocate that observes application-specific data, such as mode changes, in order to make more predictive reservation requests. The different strategies that we have implemented are described in Section 4.4.3. In general, the strategies we have implemented for periodic tasks adjust a task's reservation so that the allocated compute time is within a small threshold above the task's demand, thereby allowing the task to meet its deadlines. Other implemented advocates are appropriate for controlling aperiodic and continuous-rate tasks, for example, by requesting user-specified fixed shares of the CPU.

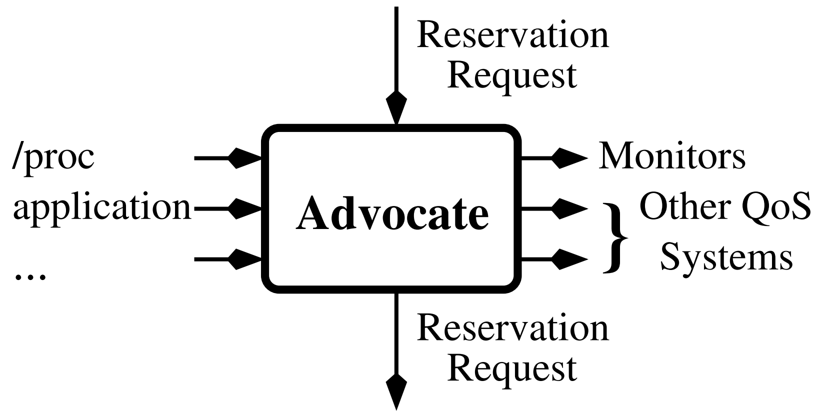


Figure 4.2. Advocate

The reservation that is requested by an advocate is called the *advice*. An advocate computes its advice and then invokes the next advocate in the chain, passing it both the task status and the computed advice.

Subsequent advocates are used to modify the advice or perform some side-effect: allowing advocates to be composed in this way greatly increases the CPU Broker's flexibility and allows existing advocates to be reused. For example, an advocate can be used to cap the resource request of a task, or alternately, ensure that the request does not fall below some minimum reserve. A side-effecting advocate might read or write information from other QoS management middleware, such as QuO. Another kind of side-effecting advocate is a "watchdog" that wakes up when some number of expected status reports have not been received from a monitored task. Reports are normally made at the end of task cycles, but this granularity may be too large, especially in the case of a task that has an unanticipated dynamic need for a greatly increased reservation. Without the intervention of a watchdog, the task might not be able to report its need until several task periods have passed—and several deadlines missed.[1] A watchdog can be used in these cases to mitigate the effects of dynamic workload increases: the watchdog wakes up, (possibly) inspects the state of its task, and requests an increased reservation on behalf of the task. In sum, composed advocates are a powerful mechanism for building complex behaviors from simple ones, for introducing and modularizing application-specific adaptations, and for conditioning the requests that are presented to the broker's contention policy object.

4.3.2 Policies

The last advocate in the chain passes the task's status and the advice to a *policy* object that is responsible for managing the requests made on behalf of all tasks. A policy has two primary roles. First, it must resolve situations in which the incoming requests cannot all be satisfied at the same time, i.e., handle cases in which there is contention for CPU resources. Second, a policy must communicate its decisions to an underlying scheduler, which implements the actual CPU reservations.

The usual input to a policy object is a reservation request from one of the advocates in the CPU Broker. In response, the policy is responsible for re-evaluating the allocation

[1]In cases where missed deadlines are unacceptable, a system designer can easily configure the CPU Broker with different advocates to ensure that critical tasks have sufficient CPU reserves to process dynamic loads without missing deadlines. The essential point is that the CPU Broker enables system designers to choose the strategies that best meet the QoS needs of their applications.

of CPU resources on the host. Conceptually, a policy recomputes the reservations for all of the broker's managed tasks in response to an input from any advocate, but in practice, this recomputation is fast and most reports do not result in changes to the existing allocations. As with advocates, the CPU Broker provides a set of policy objects, each implementing different behaviors and conflict resolution strategies. In addition, if necessary, the implementer of a real-time system can implement and deploy a custom policy object within the broker's open framework. Most policies depend on additional data in order to be useful: for instance, the broker provides policies that resolve scheduling conflicts according to user-determined task importances. Dynamic changes to these importances—signaled via a (remote) method invocation on a policy object—will cause a policy to redetermine its tasks' allocations. The broker also provides a policy that partitions tasks into groups, where each group is assigned a maximum fraction of the CPU, and contention with each group is resolved by a secondary policy. The details of these policies, how they are set up, and how they can be changed dynamically are provided in Section 4.4.

Once a policy has determined the proper CPU reservations for its task set, it invokes a *scheduler proxy* object to implement those reservations. The scheduler proxy provides a facade to the actual scheduling services of an underlying RTOS. The RTOS (not the CPU Broker) is responsible for actually implementing the schedule and guaranteeing the tasks' reservations. If the RTOS rejects a reservation that was determined by the broker's policy, then the policy is responsible for modifying its request. (In practice, our implemented policies avoid making inadmissible reservation requests.)

The policy finishes with the scheduler and finally sends information about any changed reservations back to the advocates, which send the data up the advocate chains. An advocate may use this information to inform future requests, to cooperate with other QoS management frameworks, or to signal its application to adapt to its available CPU resources, for example. Eventually, a new CPU status report is received from one of the broker's managed tasks, and the process of the advocates and policies repeats.

4.4 Implementation

In this section we describe how the design goals of the CPU Broker are met in its actual implementation. To achieve our goal of providing an open and extensible framework for dynamically managing real-time applications, we implemented the CPU Broker using CORBA. To achieve noninvasive integration with middleware-based real-time applications and other QoS management services, we used the QuO framework. Finally, to apply and demonstrate our architecture atop a commercial off-the-shelf RTOS, we implemented the broker for TimeSys Linux. The rest of this section describes how these technologies are used in our implementation, and also, the advocate and policy algorithms that we provide as part of the CPU Broker software.

4.4.1 Scheduling and Accounting

At the bottom of our implementation is TimeSys Linux [3], a commercial version of Linux with support for real-time applications. TimeSys Linux provides several features that are key to our implementation. First, the kernel implements reservation-based CPU scheduling through an abstraction called a *resource set*. A resource set may be associated with a periodic CPU reservation; zero or more threads are also associated with the resource set and draw (collectively) from its reservation. Second, the TimeSys kernel API allows a thread to manipulate resource sets and resource set associations that involve other threads, even threads in other processes. This makes it straightforward for the CPU Broker to manipulate the reservations used by its managed tasks. Third, whenever one thread spawns another, and whenever a process forks a child, the parent's association with a resource set is inherited by the child (by default). This makes it easy for the CPU Broker to manage the reservation of a task (process) as a whole, even if the task is internally multithreaded. Finally, TimeSys Linux provides high-resolution timers that measure the CPU consumption of threads and processes. These timers were essential in providing accurate reservations—and allowing high overall CPU utilization—in the real-time task loads we studied. To allow the CPU Broker to obtain high-resolution information about all of the threads in another process, we made a very small patch to the TimeSys Linux kernel to expose processes' high-resolution timers through the "`/proc/`*`pid`*`/stat`" interface.[2]

The combination of a flexible reservation-based scheduling API and high-resolution timers allowed us to implement the CPU Broker on TimeSys Linux. The architecture of the broker is general, however, and we believe that it would be straightforward to port our current CPU Broker implementation to another RTOS (e.g., HLS/Linux [16, 17]) that provides both CPU reservations and accurate accounting.

[2]Only the Linux-standard low-resolution timers (with 10 ms granularity) are exposed in the "`/proc/`*`pid`*`/stat`" interface by default. TimeSys' high-resolution counters are made available through the `getrusage` system call, but that cannot be used to inspect arbitrary processes.

4.4.2 Openness and Noninvasiveness

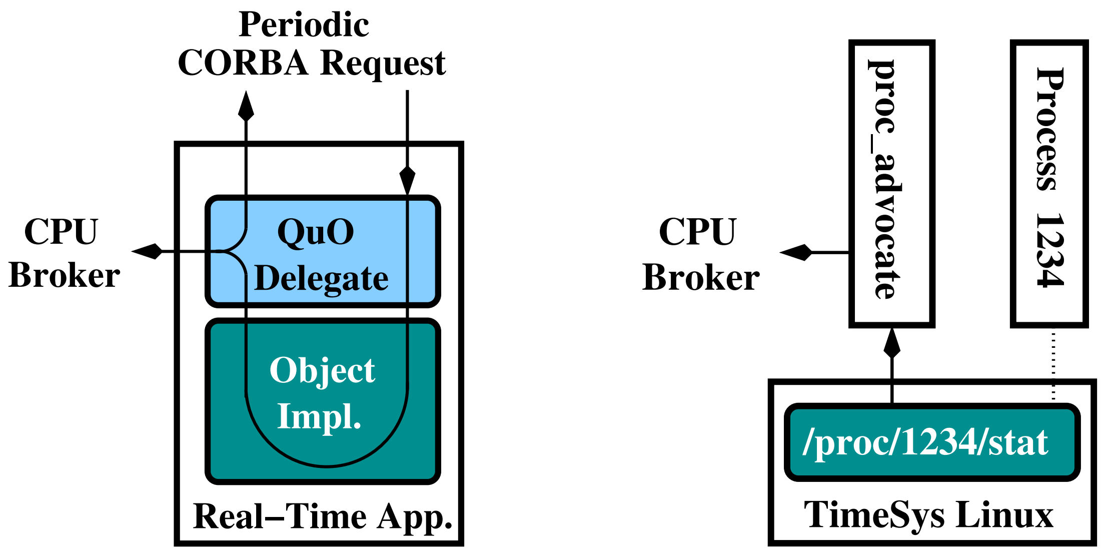
We chose to implement the CPU Broker using CORBA [1], an industry-standard middleware platform for distributed objects. CORBA provides two main features for achieving the broker's goals of openness and noninvasiveness. First, CORBA defines a standard object model and communication mechanism. By implementing the broker's advocates and policies as CORBA objects, we provide a framework for real-time systems designers to use in configuring and extending the broker. Second, CORBA abstracts over the locations of objects: communicating objects can be located in a single process, on different processes on a single machine, or on different machines. This has practical significance for both usability and performance. The broker can be easily extended with new advocates and policies without modifying existing broker code: this enables rapid prototyping, late (e.g., on-site) and dynamic customization, and cases in which a custom broker object is tightly coupled with an application (and therefore is best located in the application process). When performance is critical, new objects can be located in the same process as other broker objects; high-quality implementations of CORBA can optimize communication between colocated objects.

As described in Section 4.1, middleware in general and CORBA in particular are increasingly important for the cost-effective development of reliable real-time and embedded systems. Using CORBA in the implementation of the CPU Broker allows us to leverage this trend. We can rely on high-quality real-time middleware—in particular, the TAO [18] real-time CORBA ORB—in our implementation and also take advantage of the increasing popularity of CORBA for the development of DRE systems. More important, however, is that CORBA provides a basis for noninvasively connecting the CPU Broker to the real-time CORBA-based tasks that it manages.

A primary goal in designing and implementing the broker was to support applications that are not developed in conjunction with our system: in other words, to support programs in which the "application logic" is decoupled from the management and control of the application's real-time behavior. This makes both the applications and our CPU Broker more flexible, and it allows real-time control to be modularized within the broker rather than being scattered throughout many programs. Effective support for this programming style requires that the broker be able to integrate with its managed tasks in a noninvasive manner, i.e., in ways that require minimal or no changes to the code of the managed tasks. The broker itself runs as a user-level process that acts as the default container for the advocate, policy, and scheduler objects described previously. Integration with real-time applications therefore requires that we build "transparent bridges" between the CPU Broker and those real-time tasks. We have implemented two strategies for noninvasive integration as illustrated in Figure 4.3.

The first strategy inserts a proxy object, called a *delegate*, into a CORBA-based real-time application. This strategy is appropriate when the real-time work of an application is modularized within one or more CORBA objects, as shown in Figure 4.3(a). The broker's delegates are implemented with QuO [4], which provides a family of languages and other infrastructure for defining and deploying delegates. The implementation of the reporting delegate class is generic and reusable, not application-specific. Further, delegates can typically be inserted in ways that are transparent to the application, or localized to just the points where objects are created. A C++ or Java programmer might add just a few lines of code to a program's initialization or to a factory method [19]; alternatively, the code can be integrated in a noninvasive manner via aspect-oriented programming [20]. In our experience, delegates can often and effectively modularize the monitoring and control of real-time behavior in CORBA servers.

For applications that are not built on middleware, however, a second strategy is required. For these cases, we have implemented a "process advocate" (`proc_advocate`) as illustrated in Figure 4.3(b). The process advocate is an adapter between an unmodified application



(a) Via QuO (b) Via `proc_advocate`

Figure 4.3. Noninvasive connections between tasks and the CPU Broker

and the broker. The `proc_advocate` is a process: it requests a reservation from the broker and then forks the unmodified application as a child. The `proc_advocate`'s reservation is inherited by the child process, as described in Section 4.4.1. The `proc_advocate` is then responsible for monitoring the CPU usage of the application and reporting to the broker. Information about the child's resource usage is obtained from kernel's "`/proc/`*`pid`*`/stat`" interface, extended with high-resolution timers as explained previously. Other data about the child process, such as its period, are specified as command-line options to the process advocate.

4.4.3 Example Advocates and Policies

The CPU Broker implements an open framework for configurable and extensible control of real-time applications. To demonstrate the framework, in particular for tasks with dynamically changing CPU requirements, we implemented a core set of scheduling advocates and policies. These objects are generic and reusable in combination to construct a variety of adaptive systems, but we have not attempted to implement a complete toolbox. Rather, we have implemented example advocates and policies that we have found useful to date.

For conditioning the feedback from tasks with dynamically changing demands, the broker provides two advocates called *MaxDecay* and *Glacial*. MaxDecay tracks recent feedback from its task: every time it receives a report, a MaxDecay advocate requests a reservation that satisfies the application's greatest demand over its last n reports, for a configured value of n. A Glacial advocate, on the other hand, is used to adapt slowly: it adjusts the requested reservation by a configured fraction of the difference between its task's current reservation and the task's current demand. In general, MaxDecay advocates are useful for applications whose normal behavior includes frequent spikes in demand that should be anticipated, whereas Glacial advocates are useful for tasks whose spikes represent abnormal situations that are not useful in predicting future demands.[3] Other advocates provided by the broker include an auxiliary advocate for coordinating with QuO—the advocate sends the status and advice data to QuO "system condition" objects—and an advocate for logging data to files.

[3]Note that the distinction is based on an application's anticipated *behavior* and not its *importance*. An advocate always considers its application to be "important." It is the job of a policy, not an advocate, to make decisions based on user-assigned importance values.

The broker provides three main policy objects. The first, called *Strict*, allocates reservations to tasks in order of their user-assigned importances. Requests from high-priority tasks are satisfied before those from low-priority tasks: when there is contention for CPU time, important tasks will starve less important tasks. This policy is good for periodic tasks with critical deadlines, because important tasks can be strongly isolated from less important ones. The second policy, *Weighted*, satisfies requests according to a set of user-assigned task weights. When there is contention for resources, all reservations are reduced, but in inverse proportion to the tasks' weights: "heavy"/important tasks are less degraded than "light"/unimportant tasks. This policy implements the adaptive reservation algorithm described by Abeni and Buttazzo [6], except that the inputs to our policy are possibly modified by the advocates described above. Weighted is often preferred over Strict for aperiodic and CPU-bound tasks. The third policy provided by the broker is *Partition*, which divides the available CPU resources into two or more parts. The resources within each part are controlled by a separate policy object, i.e., a Strict or Weighted policy. This allows for isolation between task groups and combinations of contention policies, e.g., for managing both periodic and aperiodic tasks. The Partition policy keeps track of which tasks belong to which groups, and it allows an administrator to move tasks between groups at run time. By manipulating the subpolicy objects, the administrator can also dynamically change the amount of CPU time available within each subpolicy.

4.4.4 Using the CPU Broker

Finally, as a practical matter, the interface to starting and configuring the CPU Broker is critical to its use. As described previously, the broker is normally run as a single process that acts as a container for the broker's CORBA objects. When this process begins, it creates a bootstrap object that can be contacted by other CORBA tools in order to configure the broker and implement a desired CPU management strategy. We provide a command-line tool, called `cbhey`, that allows interactive communication with the broker, its advocates, and its policies. For example, setting the importance of a task named `mplayer` is as simple as using `cbhey` to talk to the controlling policy object and telling that policy to "`set priority of task mplayer to 5`." Connecting an external advocate or policy object to the CPU Broker is accomplished by using `cbhey` to give the location of the object to the broker; CORBA handles communication between the broker and the external object. Tighter and automated integration with the CPU Broker is achieved by making (remote) method invocations on the CORBA objects within the broker. For example, an end-to-end QoS management framework would likely interact with the broker not via `cbhey`, but instead by making CORBA calls to the broker's objects. For performance-critical situations, we also provide a mechanism for dynamically loading shared libraries into the main broker process.

4.5 Evaluation

Our evaluation of the CPU Broker is divided into three parts. First, we measure the important overheads within the broker's implementation and show that they are small and acceptable for the types of real-time systems we are targeting. Second, we evaluate the broker's ability to make correct CPU allocations for a set of synthetic workloads, with and without dynamically changing resource requirements, and with and without contention. Finally, we demonstrate the CPU Broker applied to a simulated DRE military application. We extend the broker with an application-specific advocate, measure the broker's ability to make appropriate CPU allocations, and evaluate the impact on the quality of service achieved by the system in the face of CPU contention.

All of our experiments were performed in Emulab [21], a highly configurable testbed for networking and distributed systems research. Each of our test machines had an 850 MHz Pentium III processor and 512 MB of RAM. Each CPU Broker host ran TimeSys Linux/NET version 3.1.214 (which is based on the Linux 2.4.7 kernel) installed atop Red Hat Linux 7.3. The CPU Broker and other CORBA-based programs were built using TAO 1.3.6 and a version of QuO (post-3.0.11) provided to us by BBN. For experiments with the distributed military application, the three hosts were connected via three unshared 100 Mbps Ethernet links.

4.5.1 Monitoring and Scheduling Overhead

There are two primary overheads in the broker: obtaining CPU data from the kernel, and communication between the broker objects via CORBA. When an application is monitored by a QuO delegate or `proc_advocate`, communication involves inter-process communication (IPC). To measure the total overhead, we built three test applications. The first was monitored by our ordinary QuO delegate, which performs two-way IPC with the broker: it sends the task status and waits to receive new reservation data. The second was monitored by a special QuO delegate that performs one-way IPC only: it does not wait for the broker to reply. The third was monitored by an "in-broker process advocate":

an advocate that functions like `proc_advocate`, but which lives in the main CPU Broker process and is run in its own thread. (The performance of an ordinary `proc_advocate` is similar to that of a two-way QuO delegate.) Each test application ran a single sleeping thread. Periodically (every 33 ms), each monitor measured the CPU usage of its task plus itself via the "`/proc/`*`pid`*`/stat`" interface and sent a report to the broker. Each test was run on an unloaded machine with a CPU Broker containing a MaxDecay advocate and a Weighted policy. We ran each test and measured the CPU and real time required for processing each of the first 1000 reports following a warm-up period.

The average times for the reports are shown in Table 4.1. The first data column shows the average of the total user and kernel time spent in both the monitor and broker per report: this includes time for obtaining CPU data, doing IPC if needed, and updating the reservation. The second column shows the average wall-clock time required for each report as viewed from the monitoring point. This is a measure of per-report latency: in the QuO delegate cases, it is less than total CPU time because it does not account for time spent in the broker. We believe that the measured overheads are reasonable and small for the class of real-time systems based on COTS operating systems and middleware that we are targeting. Increasing the number of tasks or decreasing the tasks' periods will increase the relative overhead of the broker, but when necessary, additional optimizations can be applied. For example, one might configure the monitors with artificially large reporting periods, at some cost to the broker's ability to adapt allocations quickly.

4.5.2 Synthetic Applications

To test the broker's ability to make correct CPU reservations, we used Hourglass [22] to set up two experiments. Hourglass is a synthetic and configurable real-time application: its purpose is to analyze the behavior of schedulers and scheduling systems such as the CPU Broker.

Our first experiment tests the CPU Broker's ability to track the demands of a periodic

Table 4.1. Average measured overhead of reports to the CPU Broker

Configuration	Monitor+Broker CPU Time (usec)	Monitor Only Real Time (usec)
Two-way QuO delegate	1742	1587
One-way QuO delegate	1716	660
In-broker process advocate	400	400

real-time task with a dynamically changing workload. The goal is for the reservations over time to be both adequate (allowing the task to meet its deadlines) and accurate (so as not to waste resources). We assume that the system designer does not know the shape of the application's workload over time, only that it goes through phases of increased and decreased demand. In a typical case like this, a MaxDecay advocate is appropriate for adapting to the task. There is no CPU contention in this experiment, so the policy is unimportant: we arbitrarily chose to use the Strict policy.

To drive this experiment, we created a test program that uses the core of Hourglass in a simple CORBA application. Our program's main loop periodically invokes a colocated CORBA object, which computes for a while and then returns to the main loop. The object's compute time can be fixed or variable in a configured pattern. We introduced a QuO delegate between the main loop and the object in order to connect our application to the CPU Broker. We configured our test application to have a period of 300 ms and to run a time-varying workload. The task goes through phases of low, medium, and high demand (with compute times of 100, 150, and 200 ms), and each phase lasts for 10 periods.

We ran our test application in conjunction with the CPU Broker, and the results are shown in Figure 4.4. The graph illustrates two points. First, the reservations made by the CPU Broker accurately track the demand of the application over time. Second, the MaxDecay advocate operates as intended, predicting future CPU demand based on the greatest recent demand of the task. This prevents the broker from adapting too quickly to periods of reduced demand, which is appropriate for tasks that have occasional but unpredictable periods of low activity.[4] If more accurate tracking were required, a designer would configure the advocate to observe fewer recent reports or replace the advocate altogether with a better predictor (as we do in Section 4.5.3).

The second synthetic experiment tests the CPU Broker's ability to dynamically arbitrate CPU resources between competing tasks. There are two kinds of dynamic events that require updates to the broker's reservations: changes in the requests from tasks, and changes to the policy. Because the previous experiment concerned adaptation to tasks, we chose to focus this experiment on changes to the policy.

We set up three instances of the (unmodified) Hourglass application, each connected to the CPU Broker with a "continuous rate" advocate that makes a single report describing

[4]This behavior in turn makes resource allocations more stable in multi-task systems. When there are several applications competing for resources, it is often undesirable to adjust resource allocations too frequently.

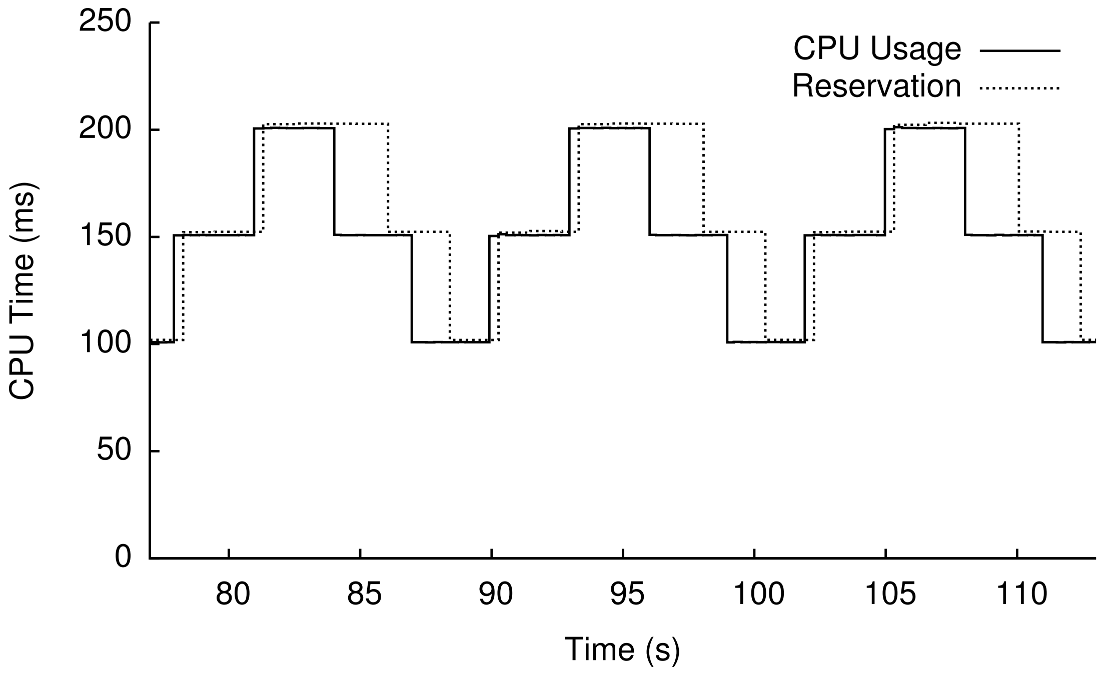


Figure 4.4. Actual compute time and reserved compute time for a task with a time-varying workload. The broker is configured to adapt to increased demands immediately, and to reduced demands only after several periods of reduced need. The MaxDecay task advocate used in this experiment can be configured to track the actual demand more or less closely.

its task's desired reservation. The compute times of the tasks were set to 95, 125, and 40 ms per 250 ms period. The broker was configured with a Strict (importance-based) policy; further, the policy was set to reserve at most 75% of the CPU to all tasks.

We then ran the three Hourglass processes and used `cbhey` to change the importances of the three tasks dynamically. The results of this experiment are shown in Figure 4.5. At the beginning of the time shown, the importance values of the three tasks are 10, 5, and 1, respectively. The Strict policy correctly satisfies the demand of task 1, because it is most important, and this task meets its deadlines. The remaining available time is given to task 2, but this reservation is insufficient and the task generally misses its deadlines, as does task 3. (These tasks can use unreserved CPU on a best-effort basis to occasionally meet their deadlines.) At time 16.6, we raise the importance of task 3 to 7, making it more important than task 2. In response, the broker reallocates CPU resources from task 2 to task 3, and task 3 begins to meet is deadlines. At time 22.5, we lower the importance of task 3 back to 1, and the broker again recomputes its reservations.

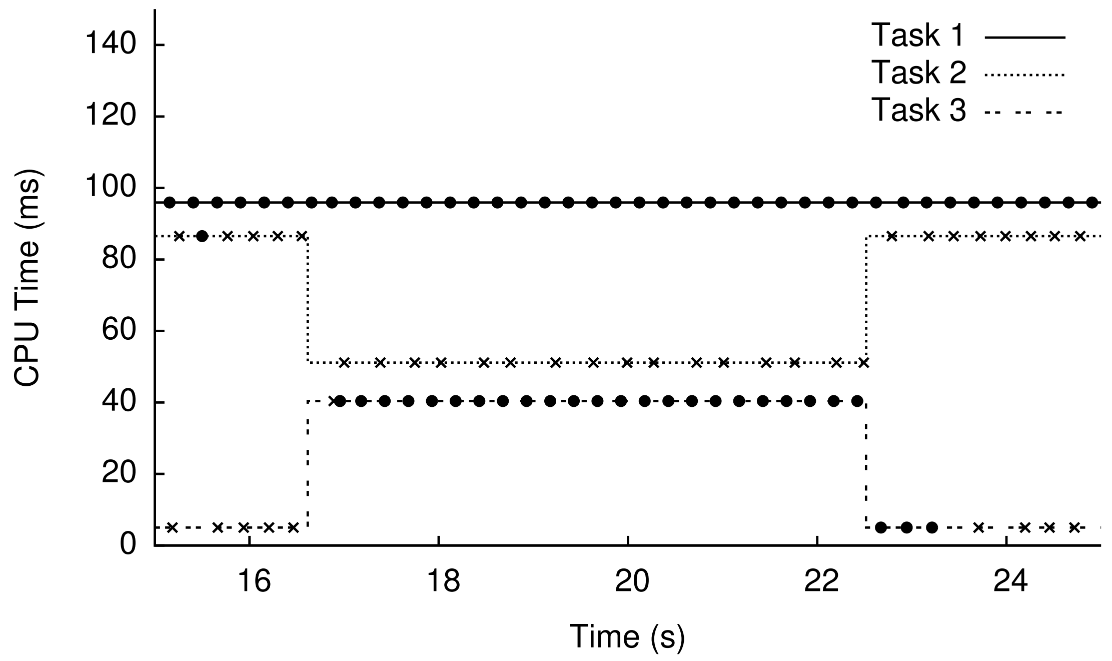


Figure 4.5. Dynamically changing reservations in response to changing task importances. Task 1 requires 95 ms every 250 ms; tasks 2 and 3 require 125 ms and 40 ms with the same period. The lines show the compute time reserved for each task. When task importances change, the broker policy updates reservations according to the Strict policy. Marks on the lines show the ends of task cycles that met (•) and did not meet (×) their deadlines.

This experiment highlights three key points. First, the configured broker policy works correctly and allocates resources to the most important tasks. As described previously, the broker implements and is open to other policies that arbitrate in different styles. Second, the broker properly and quickly adjusts reservations in response to changes to the policy configuration. In a real system, these kinds of changes could be made by an automated QoS management system or by a human operator. Third, the total utilization of the system is constantly high. In other words, the broker allocates all of the CPU time that it is allowed to assign, and the sum of the three reservations is always equal to that amount. The broker enables a system designer to keep utilization high while choosing how to divide resources in the face of contention.

4.5.3 The UAV Application

A primary goal of the CPU Broker is to provide flexible CPU control in real-time and embedded systems that are developed atop COTS middleware and that operate in highly dynamic environments. To demonstrate the benefits of the broker to these kinds of applications, we incorporated our broker into a CORBA-based DRE military simulation called the Unmanned Aerial Vehicle (UAV) Open Experimentation Platform [14]. This software simulates a system of one or more UAVs that fly over a battlefield in order to find military targets. The UAVs send images to one or more ground stations, which forward the images to endpoints such as automatic target recognition (ATR) systems. When an ATR process identifies a target, it sends an alert back to the originating UAV.

We applied our CPU Broker to the ATR stage to ensure that the ATR could reliably keep up with the flow of images coming from a UAV. The ATR task is a Java program that receives images at a rate of two frames per second. Because the ATR is CORBA-based, we used a QuO delegate to monitor the ATR and report to the CPU Broker after each image is processed. Inserting the delegate required simply adding a few lines of code to the application's main class. This change was introduced noninvasively by running the ATR with a modified class path. The Java process contains many threads, but the details of how these threads are managed by Java are unimportant to the CPU Broker. Managing the ATR relies on the broker's ability to measure the aggregate CPU consumption of all threads within a process and make a reservation that is shared by all those threads. (See Section 4.4.1.)

Because the ATR is a Java process, it periodically needs to garbage collect (GC). During these periods, the CPU demand of the ATR is much greater than its demand during normal periods. This kind of situation is not uncommon in software that was not carefully designed for predictable behavior. We could have dealt with this problem by configuring the CPU Broker with a MaxDecay advocate which keeps the ATR's reservation high in anticipation of future GC cycles, but this would have been unnecessarily wasteful. Instead, we implemented a custom advocate that predicts when the ATR will GC and requests an increased CPU reservation only in those cases—a *proactive*, rather than a *reactive* advocate.

The behavior of the ATR and our custom advocate are shown in Figure 4.6. The graph shows the periodic demand of the ATR and the predictions made for the ATR by our advocate. These lines often match closely, but our advocate is not a perfect predictor. Still, the forecast from our custom advocate is better than we could achieve with MaxDecay, which would consistently over-allocate or be slow to react to spikes.

We then ran the UAV software with and without the CPU Broker to test the broker's ability to improve the run-time behavior of the system under CPU load. We used three machines running one UAV process, one distributor, and one ATR process, respectively.

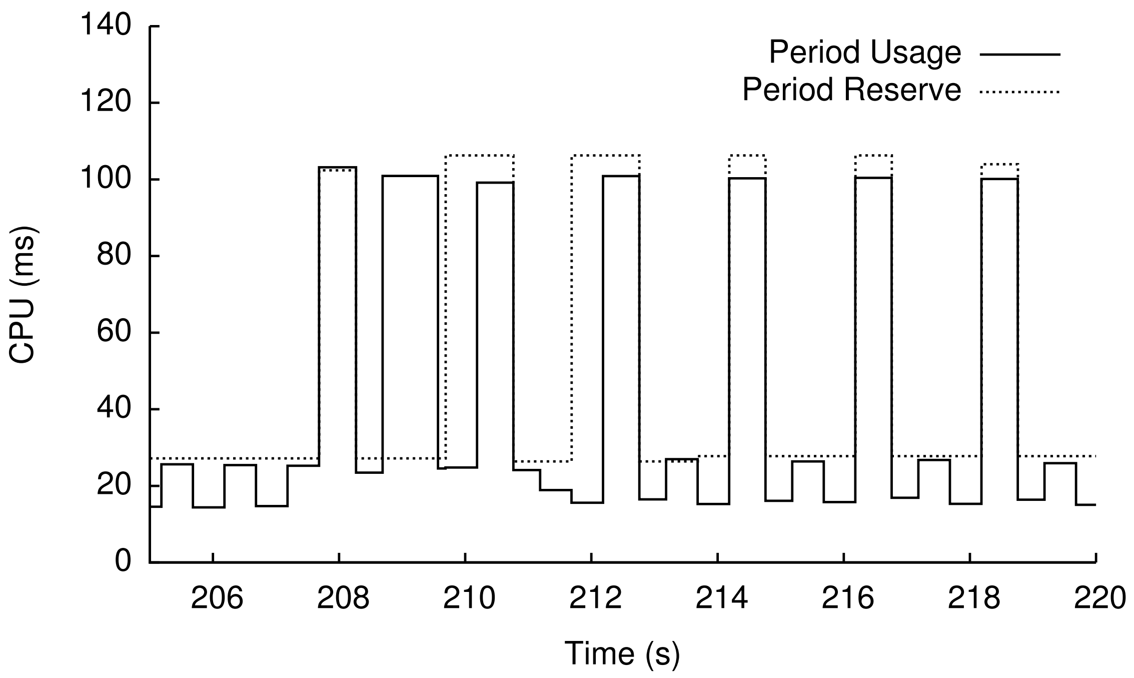


Figure 4.6. Comparison of the actual compute time and reserved compute time for the ATR. The ATR's demand has regular spikes that correspond to garbage collections. The custom advocate predicts and adapts to this behavior.

The third machine also ran a simple process that receives images from the distributor and sends them to the ATR. We ran this simulation in three configurations, for 220 seconds each time, and collected data about the reliability of the system.

The results of our tests are shown in Table 4.2. We first ran the UAV software without introducing the CPU Broker or any competing CPU loads in order to obtain baseline measures. The table shows the rate of image processing (measured at the receiver over 3-second windows), the latency of alerts (delay seen by the UAV between its sending of a target image and its receipt of the corresponding alert from the ATR), and the total numbers of images and alerts processed by the system. We then added a competing real-time task on the ATR host—an Hourglass task with a reservation for 90 ms every 100 ms—and ran the simulation again. The results in the second column show that the system behaves unreliably: many images and alerts are lost. Finally, we used the CPU Broker on the ATR host in order to prioritize the receiver and ATR processes above Hourglass. The third column shows that image handling in the broker-managed UAV system is similar to that in the system without load. Similarly, no alerts are lost, but their latencies are increased for two reasons. First, our advocate occasionally mispredicts GC cycles: we could use a different advocate to improve reliability, at a cost in overall system utility. Second, although the ATR receives its reservation, the RTOS may spread the compute time over the full period (500 ms), thus increasing alert latency. We could address this problem in the future by adding deadline information to our CPU Broker interfaces. In sum, our experience showed that the broker can noninvasively integrate with a CORBA-based DRE system and improve that system's reliability in the face of CPU contention.

Table 4.2. Performance of the UAV simulation

Metric	**Unloaded, Baseline**	**CPU Load**	**CPU Load, With Broker**
Frames processed	432	320	432
Avg. frames per second	1.84	1.32	1.81
Min. frames per second	1.67	0.45	1.11
Max. frames per second	2.00	2.01	1.99
Std. Dev.	0.09	0.34	0.09
Alerts received	76	50	76
Avg. latency (ms)	127.67	1560.44	325.72
Min. latency (ms)	101.00	362.00	145.00
Max. latency (ms)	193.00	3478.00	933.00
Std. Dev.	33.46	961.62	153.60

4.6 Conclusion

Embedded and real-time systems are increasingly dependent on the use of COTS infrastructure and the reuse of software parts—even entire applications. Furthermore, systems are increasingly deployed in environments that have changing sets of computing resources and software with dynamically changing requirements. We have presented the design and implementation of our CPU Broker that addresses the needs of these systems in an open and extensible fashion. Our architecture supports adaptive, feedback-driven CPU reservations and explicitly separates per-task and global adaptation strategies. Our implementation atop a commercial RTOS effectively determines and adapts CPU reservations to the dynamic requirements of its managed tasks, with low overhead. Finally, the broker effectively modularizes the strategy for allocation and adaptation, and connects to both middleware-based and other applications in a nonintrusive manner. In conclusion, we believe that the broker approach can provide important benefits toward achieving understandable, predictable, and reliable real-time behavior in a growing and important class of real-time and embedded software systems.

4.7 Availability

The CPU Broker is open-source software. Complete source code and documentation for the CPU Broker are available at `http://www.cs.utah.edu/flux/alchemy/`.

4.8 Acknowledgments

We thank Craig Rodrigues at BBN for his continual and ongoing help with the UAV software. We also thank Alastair Reid for his valuable comments on drafts of this paper. Finally, we thank Leigh Stoller, Rob Ricci, Mike Hibler, Kirk Webb, and other members of the Emulab support staff for helping us deploy and evaluate the CPU Broker in that facility.

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

4.9 References

[1] Object Management Group, *Common Object Request Broker Architecture: Core Specification*, Dec. 2002, revision 3.0.2. OMG document formal/02–12–06.

[2] ——, *Real-Time CORBA Specification*, Nov. 2003, version 2.0. OMG document formal/03–11–01.

[3] TimeSys Corporation, "TimeSys Linux GPL: Performance advantages for embedded systems," 2003, white paper, version 1.1.

[4] J. A. Zinky, D. E. Bakken, and R. D. Schantz, "Architectural support for quality of service for CORBA objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 55–73, 1997.

[5] J. Regehr and J. Lepreau, "The case for using middleware to manage diverse soft real-time schedulers," in *Proceedings of the International Workshop on Multimedia Middleware (M3W '01)*, Ottawa, ON, Oct. 2001, pp. 23–27.

[6] L. Abeni and G. Buttazzo, "Adaptive bandwidth reservation for multimedia computing," in *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, Hong Kong, China, Dec. 1999, pp. 70–77.

[7] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a reservation-based feedback scheduler," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS '02)*, Austin, TX, Dec. 2002, pp. 71–80.

[8] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium (RTAS '99)*, Vancouver, BC, Jun. 1999, pp. 111–120.

[9] T. Nakajima, "Resource reservation for adaptive QOS mapping in Real-Time Mach," in *Parallel and Distributed Processing: 10th International IPPS/SPDP '98 Workshops*, ser. Lecture Notes in Computer Science, J. Rolim, Ed. Springer, Mar.–Apr. 1998, vol. 1388, pp. 1047–1056, in the *Joint Workshop on Parallel and Distributed Real-Time Systems*.

[10] Object Management Group, *Real-Time CORBA Specification*, Aug. 2002, version 1.1. OMG document formal/02–08–02.

[11] C. Lu, X. Wang, and C. Gill, "Feedback control real-time scheduling in ORB middleware," in *Proceedings of the Ninth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '03)*, Washington, DC, May 2003, pp. 37–48.

[12] S. Brandt, G. Nutt, T. Berk, and J. Mankovich, "A dynamic quality of service middleware agent for mediating application resource usage," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS '98)*, Madrid, Spain, Dec. 1998, pp. 307–317.

[13] L. Abeni and G. Buttazzo, "Hierarchical QoS management for time sensitive applications," in *Proceedings of the Seventh IEEE Real-Time Technology and Applications Symposium (RTAS '01)*, Taipei, Taiwan, May–Jun. 2001, pp. 63–72.

[14] D. A. Karr, C. Rodrigues, J. P. Loyall, R. E. Schantz, Y. Krishnamurthy, I. Pyarali, and D. C. Schmidt, "Application of the QuO quality-of-service framework to a distributed video application," in *Proceedings of the Third International Symposium on Distributed Objects and Applications (DOA '01)*, Rome, Italy, Sep. 2001, pp. 299–308.

[15] R. E. Schantz, J. P. Loyall, C. Rodrigues, and D. C. Schmidt, “Controlling quality-of-service in a distributed real-time and embedded multimedia application via adaptive middleware,” Jan. 2004, submitted for publication, http://www.cs.wustl.edu/~schmidt/PDF/D&T.pdf.

[16] L. Abeni, “HLS on Linux,” Nov. 2002, http://hartik.sssup.it/~luca/hls/.

[17] J. Regehr and J. A. Stankovic, “HLS: A framework for composing soft real-time schedulers,” in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS ’01)*, London, UK, Dec. 2001, pp. 3–14.

[18] D. C. Schmidt, D. L. Levine, and S. Mungee, “The design of the TAO real-time object request broker,” *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

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

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *ECOOP ’97: Object-Oriented Programming*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds. Springer, Jun. 1997, vol. 1241.

[21] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, “An integrated experimental environment for distributed systems and networks,” in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI ’02)*, Boston, MA, Dec. 2002, pp. 255–270.

[22] J. Regehr, “Inferring scheduling behavior with Hourglass,” in *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, Monterey, CA, Jun. 2002, pp. 143–156.

PART IV

INFLUENCE AND CONCLUSION

CHAPTER 5

INFLUENCE

The work presented in Chapters 2 through 4 was published in three papers and presented at peer-reviewed conferences. This chapter summarizes the apparent influence of those publications on subsequent related efforts. Its purpose is to illustrate how the novel realization of variability mechanisms, implemented so as to improve the run-time performance of software, continues to be relevant to the field of software engineering. This chapter focuses on, but is not limited to, published work that references the conference papers that are reprinted in Chapters 2 through 4.

5.1 Flexible and Optimizing IDL Compilation

Flick incorporated techniques from traditional programming-language compilers to bring flexibility and optimization, at the same time, to the domain of IDL compilation. Whereas ordinary IDL compilers were designed to implement a single IDL mapping for a single message-transport system, Flick was implemented as a "kit" of components that supported multiple IDLs, IDL mappings, and message transports. Whereas ordinary IDL compilers did little to produce optimized stubs, Flick used a family of intermediate code representations that admitted compile-time optimizations for stub performance. These two aspects made it possible for Flick to generate specialized and optimized RMI code for Fluke [37], an experimental microkernel developed by the Flux Research Group at the University of Utah, in addition to the other standards that Flick supported.

Flick was influential in all of these areas. Flick affected subsequent research and development efforts toward flexible IDL compilers, which generally sought to produce stubs and middleware runtime libraries that were customized to the needs of selected applications. Flick also influenced efforts toward IDL compilers and middleware that optimize for run-time performance. Finally, Flick was a catalyst for a series of IDL compilers that produce RPC code for the L4 microkernel family. The L4 work is notable for seeking an IDL compiler that produces *extremely* efficient stubs, but which also supports easy modification of the IDL compiler to accommodate continual changes to L4.

The following sections organize Flick-related work into these areas: work toward flexible compilation (Section 5.1.1), work that targets run-time optimizations (Section 5.1.2), and work in the development of IDL compilers for L4 (Section 5.1.3). The literature survey in these sections suggests that, while flexibility and optimization have continued to be pursued individually, Flick remains unique in its use of rich intermediate code representations that allow an IDL compiler to address flexibility and optimization goals simultaneously.

5.1.1 Flexible IDL Compilation

Chapter 2 describes how Flick can be extended to support nonstandard mappings from the constructs of an IDL to the constructs of a programming language such as C. The actual Flick implementation described in Chapter 2, however, provided only the standard translations from CORBA IDL, ONC RPC IDL, and MIG IDL onto C. Eide et al. [24, 25] subsequently demonstrated Flick's ability to support nonstandard translations by implementing a new mapping from CORBA IDL onto "decomposed" client and server stubs, which were designed to support applications that needed finer-grain control over communication events.

Using the normal presentation style, Flick maps IDL interfaces onto client stubs and server skeletons that encapsulate many steps of RPC or RMI communication. For example, each client stub encapsulates the steps of marshaling a request, sending the request, waiting for the reply, and unmarshaling the response. In the decomposed presentation style, Flick creates individual stubs for each of these steps. As Eide et al. describe [25, page 280], the decomposed presentation style consists of "*marshaling stubs*, to encode request and reply messages; *unmarshaling stubs*, to decode request and replay messages; *send stubs*, to transmit marshaled messages to other nodes; *server work functions*, to handle received requests; *client work functions*, to handle received replies; and *continuation stubs*, to postpone the processing of messages." Eide and his colleagues modified a distributed application based on asynchronous message passing so that it used the decomposed stubs. The custom mapping made it possible for the application's interfaces to be defined by CORBA IDL, and the Flick-generated stubs replaced the application's previously hand-coded communication substrate.

Eide et al. observed that the decomposed stubs supported communication patterns that could not have been accommodated by the standard CORBA IDL mapping. In particular, it allowed a server to premarshal common responses, and it allowed a distributed set of servers to make multiple replies to a single request.[1] Eide also noted that in the decomposed style, local-object optimizations became the responsibility of the application: a client should invoke a marshaling stub only when the target object is remote. To move local-object optimizations back into the middleware, Eide and his colleagues suggested that decomposed stubs could postpone actual request-message marshaling until the request is sent to a remote object.

To significantly change the data types and functions output by Flick, a developer must implement a new presentation generator—a nontrivial task, but one that is eased by Flick's substantial libraries. Auerbach et al. [11] explored a different approach to the flexible creation of stubs. Their Mockingbird tool creates stubs from pairs of interface declarations that define the endpoints of an RPC channel. A programmer provides the datatypes and function signatures that he or she wants to use at the client and server, and Mockingbird produces code that implements the connection. This declarative style provides obvious advantages: it is easy for users to understand, it requires little specification effort from users, and it allows Mockingbird to produce stubs that refer to existing datatypes in the client and server programs.

Mockingbird works by deriving a structural correspondence between the two interfaces. It translates each of the endpoint definitions into an internal representation, called *Mtypes*, and then applies rules to find a unique isomorphism between the endpoints' Mtypes. The success of this search may depend on type annotations that express constraints: for example, there are annotations for stating that a field is non-null, that fields do not alias, that one field gives the length of another, and that a generic container is constrained to hold instances of a particular type. Mockingbird provides a GUI that allows users to annotate types as necessary and produce stubs.

Mockingbird allows a user to control RPC-endpoint interfaces in a direct, lightweight, and flexible manner. Its approach is not incompatible with Flick's: indeed, Mockingbird's Mtypes and annotations are comparable to Flick's MINT and PRES_C intermediate representations, and one can imagine a Flick presentation generator that would input the (annotated) signatures of the stubs that a user desires. Because Mockingbird and Flick have

[1]The application viewed each reply message as a partial response to its original request. The application had to inform the middleware ORB that it was expecting multiple replies.

comparable internal representations, it seems that it would be possible for Mockingbird to apply Flick-like code optimizations to the stubs that it creates. (Auerbach et al. described stub-code optimizations in Mockingbird as future work [11].)

It would be challenging, however, to adopt stub signatures as the *only* means of RPC specification for large systems. Even for a small IDL specification, an IDL compiler may produce quite a lot of code, and not all of it is directly related to RPC. More significantly, IDL specifications control certain details of messaging, such as the order in which the parameters of an operation are marshaled. A shared IDL specification thus ensures compatibility between many clients and servers, whereas Mockingbird only addresses compatibility between two endpoints.

Perhaps for these reasons, a more popular approach to flexible IDL compilation is through extensible compilers. A common thread in much of this work is that the extension interface allows a developer to add or tailor a compiler pass that walks over an abstract syntax tree (AST) that represents an input IDL specification. The AST is the primary internal representation used by the compiler, and the extension is tasked with producing code as it walks over the AST. Although this approach can and has been used to tailor the code produced by IDL compilers, Flick avoided it because ASTs are poor bases on which to implement the program analyses and optimizations described in Section 2.3.

The *omniidl* compiler exemplifies the AST-based approach to customizable IDL translation. It is the IDL compiler included with omniORB [50], a popular and open-source implementation of the CORBA middleware standard. As described in its manual [49], omniidl allows developers to write a custom back ends for the compiler. A back end is expected to walk over the AST representation of an IDL specification, using the *Visitor* design pattern [39], and produce output. The omniidl manual describes utility libraries that make it easier for back ends to manage streams and produce output based on code templates. It does not, however, make any mention of support for any kind of code optimization, and indeed, the C++ back end provided with omniidl does not implement stub-code optimizations like those found in Flick.

Even without optimizations in the generated code, it can be useful to tailor an IDL compiler to the requirements of particular applications. Welling and Ott describe their experience in implementing a configurable, template-driven IDL compiler and applying it to create tailored stubs and skeletons [102]. Similar to omniidl, their compiler was implemented in two parts: (1) a front end that parses CORBA IDL and produces an abstract syntax tree, and (2) a back end that inputs the AST and a set of templates, and from those produces and stubs and skeletons. Using their new compiler's extensibility, they implemented a special back end to produce C++ code for HEIDI, a large and existing application for testing multimedia software. The back end produced stubs that incorporated HEIDI's existing datatypes, and internally, the stubs utilized HEIDI's library for inter-object messaging. They suggest that template-driven IDL compilers can be useful in general, whenever it is beneficial to adapt middleware to particular applications, as opposed to adapting the applications to standard middleware interfaces. Welling and Ott also suggest that the strengths of their approach are complementary to those of Flick, because while their templates do not easily support optimizations during code generation, it is easier for a developer to write a template than a new Flick component. They suggest [102, page 412]: "A good strategy may be to utilize the template approach when code-generation flexibility is desired, but resort to writing a custom Flick back-end for incorporating sophisticated optimizations."

IDLflex is another compiler that achieves code-generation flexibility through a template-driven back end [83]. Implemented by Reiser et al. for use in the AspectIX middleware system [53], IDLflex is structurally similar to the two compilers discussed above. The front end, which is not designed for extensibility, parses a CORBA IDL specification into an internal AST. Subsequently, the back end processes a template to translate the AST representation into stubs and skeletons. Reiser and his colleagues state that this architecture made it simple for them to experiment with code generation for AspectIX. In comparison to Flick, "which urges the programmer to write C++ code for complex mapping changes" [83, page 8], Reiser et al. state that changing the output of IDLflex is an easy and rapid process. The IDLflex template language is a custom, XML-based programming language designed for producing text; it is not well suited to performing code optimizations like those that Flick performs.[2]

Swint et al. implemented another flexible IDL compiler that relies on XML: the Infopipe Stub Generator (ISG), which was built using the "Clearwater approach" to code generation. The dual foundations of the Clearwater approach are, first, to use XML as the basis of a compiler's intermediate representations, and second, to use XSLT (Extensible Stylesheet

[2]Subsequent to the work described in Chapter 2, Tim Stack implemented an XML-based and template-driven code-generation system for Flick [35]. The template-driven system was used to produce "boilerplate" code, primarily for the CORBA C++ language mapping. It complemented the CAST-based output system, described in Chapter 2, which was used to produce optimized stubs and skeletons.

Language Transformations) to implement the compiler's passes. Unlike IDLflex, which transforms an XML template into target code in a single pass, the output of a Clearwater pass is generally an XML document with target code embedded: i.e., annotated or "marked up" code. The XML markup identifies semantically meaningful points in the generated code. Because a Clearwater compiler pass both consumes and outputs an XML document, it is straightforward to implement a complete compiler as a pipeline of XML transformations. The last stage of the pipeline removes all remaining XML markup, leaving only the final, generated code.

Swint and his colleagues identified the advantages their XML-based approach. First, it supports extensibility. Because XML is generic, it is suitable as an intermediate representation for compilers of many languages (not just IDL compilers, for example). XML's genericity also makes it trivial for a compiler author to define the XML tags that he or she needs to implement any particular compiler. Second, XML and XSLT support flexibility. XSLT and its related languages, including XPath, allow programmers to write transformations are that robust in the face of many structural changes to the XML formats on which they operate. Thus, compiler developers can often make changes to the intermediate representations of programs without needing to make compensatory changes in their XSLT-defined compiler passes. Third, Clearwater supports modularity. Separate code-generation steps can be easily organized as separate compiler passes. To the extent that separate passes operate on different parts of an XML tree, they can be independent. In addition, the pipeline approach allows a developer to insert new passes—perhaps application-specific transformations—into an existing compiler in a modular way.

The Infopipe Stub Generator is implemented using the Clearwater approach. From an Infopipe specification (written in XIP, and XML-based language), ISG produces C and C++ implementations of the stubs needed to implement the Infopipe, and well as Makefiles for directing compilation. Beyond leveraging the Clearwater framework itself, ISG exposes Clearwater's flexibility to users by supporting stub customization through aspect-oriented programming [28]. The ISG code-generation process leaves XML tags in its output; these tags identify meaningful "join points" in the generated code. A pass called the AXpect weaver then runs to apply user-directed transformations to the code. Swint et al. state that they found this to be an effective way to manage quality-of-service concerns in the generated code. After the AXpect weaver runs, the last stage of the compiler removes remaining XML markup.

In comparing the Clearwater approach to Flick, Swint et al. state that their code generator "adopts an intermediate format for flexibility like `gcc` [the GNU Compiler Collection] and Flick" [96, page 152]. Although it is true that the intermediate representations of Clearwater and Flick are both designed to support flexibility, they are quite different in their details. Clearwater's use of XML achieves independence from the concrete input and output languages that it processes; indeed, the Clearwater approach is not even specific to IDL compilation (although any compiler pass implemented atop Clearwater would be necessarily specific to some aspects of the intermediate representation that it processes). In contrast, Flick's intermediate representations were designed to provide flexibility *only within the domain of IDL compilation.* An advantage of Flick's intermediate representations is that they are designed to support stub-code optimization. Clearwater's intermediate representations and transformation passes are essentially template-based, and as discussed previously, template processing is not well suited to the kinds of optimizations that Flick performs. In discussing run-time performance, Swint et al. state that "our Clearwater architecture poses no inherent limit on the generated code when compared to a traditional generation tool like `rpcgen`" [96, page 151]. As explained in Section 2.4, Flick-generated stubs have significantly greater marshal throughput than those produced by `rpcgen`.

The Infopipe Stub Generator supports flexible code generation through aspect-oriented programming: an ISG user controls the set of aspects that should be "woven" into the code produced by the stub compiler. Zhang and Jacobsen explored a similar approach to flexibility [106]. They performed an aspect-oriented refactorization of ORBacus, an open-source implementation of CORBA, to make it possible for an application developer to tailor ORBacus to the needs of the application: not just tuning the code produced by ORBacus' IDL compiler, but also the code within the middleware runtime. They identified three significant features of the CORBA middleware and modularized these into aspects, using the AspectJ programming language. They also made the IDL compiler aspect-aware by implementing two features. The first, called *API splitting*, allowed a user to state that certain parts of an input IDL interface should be implemented by an AspectJ aspect. In effect, this allowed a user to flexibly subset the translation of an IDL interface. The second IDL compiler feature was an option to exclude local-object optimizations from the generated stubs. Both features were implemented as aspects that are applied to the IDL compiler itself.

The flexibility provided by Zhang and Jacobsen's approach is essentially feature subsetting. This is useful but different from the type of flexibility enabled by Flick: Flick's intermediate representations are designed to support multiple IDLs, presentation styles, and message transport systems, not just the subsetting of a single "base" mapping from IDL to stubs. Flick is also designed to support optimization of the generated code. Feature subsetting can lead to performance improvements in principle, but the aspects implemented by Zhang and Jacobsen do not relate to message marshaling, and their measurements show that their aspects have little or no effect on the run-time performance of their generated stubs and skeletons [106]. Nevertheless, their aspect-oriented approach seeks to tailor both the IDL-generated code and the middleware runtime library. Flick, on the other hand, does not address flexible configuration of the middleware runtime. Zhang and Jacobsen also used aspect-oriented programming to introduce features into their IDL compiler—a relatively lightweight approach to extensions.[3] Flick, conversely, is designed for heavier-weight extensions. To extend Flick, a compiler developer is expected to write a new module, albeit atop Flick's significant libraries.

The *Abacus* system by Zhang et al. was a more complete aspect-oriented refactoring of ORBacus [105], with more middleware features and modularized within aspects. Abacus' configuration system, called *Arachne*, included an expanded version of their compiler described previously. Compared to their previous work, a major improvement in Abacus and Arachne was the largely automatic handling of application-required features and those features' dependencies. Experimental results showed that Abacus could synthesize middleware that met applications' requirements and that the synthesized middleware was superior to uncustomized middleware in terms of static code size and dynamic memory use. The optimization of these metrics is unaddressed by Flick's existing components, and as previously noted, Zhang's techniques customize the middleware runtime library whereas Flick does not. Experimental results [105, 107] also showed that Abacus' approach to synthesis—feature subsetting—continued to have little to no effect on the time required to perform client-server RMI. In summary, Abacus and Flick address different but complementary goals, both in terms of middleware flexibility and middleware optimization.

[3]Zhang and Jacobsen do not, however, provide evidence that the IDL compiler they used was designed with aspect-oriented extension in mind. They used the IDL compiler from JacORB, another open-source implementation of CORBA, in their study.

5.1.2 Stub and Skeleton Optimization

The work described in the preceding section focuses on flexible or extensible IDL compilation. Flick also influenced work in optimizing compilation, i.e., techniques for producing stubs and skeletons that execute quickly at run time.

For example, Gokhale and Schmidt [46, 47] cite Flick as an influence in their efforts to create an optimizing IDL compiler for TAO, a popular and open-source implementation of CORBA. The TAO IDL compiler generates stubs and skeletons that use an IIOP marshaling library, which Gokhale and Schmidt optimized according to seven optimization principle patterns [47, page 1678]: "1) optimizing for the common case; 2) eliminating gratuitous waste; 3) replacing general-purpose methods with efficient special-purpose ones; 4) precomputing values, if possible; 5) storing redundant state to speed up expensive operations; 6) passing information between [layers]; 7) optimizing for processor cache affinity." Whereas Gokhale and Schmidt implemented these optimizations by hand in their IIOP library, Flick implements most of them as part of the compile-time analyses it performs. For example, Gokhale and Schmidt optimize for the common case by inlining method calls. Flick performs inlining as part of code generation (Section 2.3.3). Similarly, optimizations 2 through 4 are immediate consequences of the fact that Flick produces specialized marshaling code in every stub, as opposed to producing stubs that rely on a message-marshaling library. Optimization 5 refers to keeping state in TAO's IIOP library, such as the marshaled sizes of structures; Flick determines such values at compile time. Flick also performs optimization 6 at compile time. A Flick back end deals with the entire process of marshaling or unmarshaling a message and, in so doing, performs inter-layer optimizations.

Only optimization 7, careful cache utilization, was addressed by Gokhale and Schmidt and not by Flick. Gokhale and Schmidt improved instruction-cache behavior by manually restructuring large functions in their IIOP engine into smaller functions and by generating stubs that use interpretive rather than compiled marshaling. These optimizations are particularly important on the embedded platforms that were the target of Gokhale and Schmidt's work [46, 47]. Flick's existing back ends were not designed with embedded targets in mind; its architecture, however, would allow a developer to implement a new back end to create stubs tailored for embedded systems. This would allow Flick to address the cache concerns that were addressed by the TAO IDL compiler. In the other direction, Gokhale and Schmidt stated that their future work included extending the TAO IDL compiler to produce compiled stubs based on the optimizations provided by Flick [46, 47].

Gokhale and Schmidt achieved performance improvements by carefully engineering the libraries used by IDL-derived code. Flick eschews clever runtime libraries and instead implements domain-specific optimizations at compile time (Section 2.3). Muller et al. explored a third approach to RPC performance optimization. They applied partial evaluation—a general-purpose optimization technique, rather than a domain-specific one—to the stubs produced by an IDL compiler *and* the runtime library used by those stubs [71, 72].

Muller and his colleagues used the Tempo partial evaluator [15] to statically optimize stubs and skeletons produced by `rpcgen` (the standard Sun RPC compiler) along with the library code used by those stubs and skeletons. The structure of the standard Sun RPC implementation consists of several thin layers: Tempo was able to track the propagation of known values across these layers. By performing function inlining across layers and context-specific specialization of the inlined function instances, Tempo was able to eliminate numerous sources of run-time overhead and yield optimized Sun RPC stubs and skeletons. Muller et al. summarize the benefits they obtained [71, page 240]: "In our experiment, the optimized Sun RPC runs up to 1.5 times faster than the original Sun RPC. In addition, the specialized marshaling process runs up to 3.75 times faster than the original one."

A central advantage of partial specialization is that it is general: Muller et al. demonstrated that it is possible to optimize IDL-derived code and RPC libraries automatically and without incorporating an optimizer into the IDL compiler itself. On the other hand, they noted that not all optimizations are within the reach of partial evaluation. They stated that the structure of the Sun RPC stubs and libraries hid some optimization opportunities from Tempo, and in some cases, Muller et al. modified Sun RPC code so that Tempo could specialize it. Of Flick in particular, they stated [71, page 247]: "Due to its flexible internal architecture, [Flick] can match the characteristics of the target machines and implement aggressive optimizations that [go] beyond the scope of partial evaluation." Their experiment suggests that general-purpose techniques, such as partial evaluation, and domain-specific techniques are both useful for optimizing IDL-derived code. Moreover, each can address concerns that are outside the scope of the other. For example, an IDL compiler can produce code in different styles to meet users' requirements; in Flick's case, this includes flexible stub presentation styles. A partial evaluator can then optimize the generated stubs and skeletons in conjunction with other program code, i.e., code that is outside the IDL compiler's control and view.

Muller et al. used Tempo to optimize stubs and skeletons that were produced statically, i.e., before the communicating client and server programs executed. Kono and Masuda used Tempo to specialize stubs that were created dynamically [61]. Kono and Masuda implemented an RMI framework in which senders encode messages in the native object formats of the receivers. This is a potential performance optimization: in comparison to systems such as CORBA and ONC RPC, in which the sender and receiver must each convert between its native data format and a platform-neutral message format, Kono and Masuda's system avoids one time-consuming conversion step. For this to yield a performance improvement, the sender's marshaling code must be fast, even though the receiver's data format is not known until run time. Kono and Masuda address this problem by using Tempo to optimize marshaling routines that the sender produces at run time, after the receiver's data format is known.[4]

In Kono and Masuda's system, client-server interfaces are defined in an IDL. For each IDL-defined data type, the IDL compiler produces a generic marshaling routine that takes a *layout description* as a parameter: the stub marshals an instance of the data type into a buffer according to the layout description. (Layout descriptions are defined in a language that is similar to Flick's MINT IR, with some annotations similar to those described by Flick's PRES_C IR.) At run time, to initiate communication across an IDL-defined interface, a client and server exchange layout descriptions: each tells the other about its native representations of the data types relevant to the interface. Each then invokes Tempo to produce optimized marshaling code: at each side, Tempo partially evaluates the generic marshaling routines with respect to the other side's layout description. The result is two sets of dynamically compiled stubs, one in the client and one in the server, each side specialized for communication with other side. Kono and Masuda report that in their

[4]Additional problems must be overcome in order for dynamically compiled stubs to yield performance improvements in practice. Most notably, the cost of dynamic compilation is run-time overhead. For dynamic compilation to yield an actual performance increase, the total benefit of the dynamically produced stubs must be greater than the start-up cost of producing them; achieving this often means that the stubs must be invoked many times so that the cost of dynamic compilation can be amortized over many stub invocations. Kono and Masuda do not discuss this issue [61]. They also do not discuss the problems that stem from requiring a sender to keep multiple versions of a single stub, one per receiver data format. They also do not discuss the fact that sender-makes-right may not eliminate data copying at the receiver. Whereas Kono and Masuda apparently focus on message data formats only, Flick's analysis for eliminating data copying at the receiver is based on the format of the encoded data *and* the presentation of that data, e.g., rules about the allocation and life cycles of objects at the receiver (Section 2.3.1).

experiments, the marshal throughput of their specialized stubs was 1.9 to 3.0 times greater than the marshal throughput of equivalent stubs that use the Sun XDR library [61].

With respect to Flick, Kono and Masuda [61, page 315] asserted that Flick's optimization techniques "will be able to be accommodated to our generic serializers. The more efficient a generic serializer is, the more efficient the specialized serializing routine is." This restates the claim made previously in this section that both general-purpose and domain-specific techniques are useful for optimizing IDL-derived code: Flick-like compilers and partial evaluators have complementary strengths.

There is ongoing interest in general-purpose middleware that is both flexible and optimizing. Zalila et al., for example, described their efforts [104] to implement an optimizing IDL compiler for PolyORB [101], a "schizophrenic" middleware platform for Ada programs. PolyORB supports multiple application-development standards including CORBA, DSA (the Distributed Systems Annex of Ada 95), and MOMA (Message-Oriented Middleware for Ada); it also supports multiple protocol standards including GIOP and SOAP. Moreover, PolyORB allows these front-end and back-end personalities to be paired in different ways, and it supports interoperability between applications that are written according to different standards. Such mix-and-match, run-time interoperability goes beyond what Flick sought to provide.

PolyORB's standard front and back ends communicate through a *Neutral Core Middleware* layer, in which data are encapsulated in self-describing units (the equivalent of CORBA's `Any` type). This results in high run-time overhead: in every round-tip RPC, there are eight points at which data are converted to or from the neutral intermediate format. Zalila implemented a new IDL compiler for PolyORB, called IAC, that avoids this overhead. For CORBA-derived interfaces, IAC can produce stubs that target GIOP messaging directly. IAC implements some optimizations that Flick does not; notably, it produces skeletons that use perfect hashing to dispatch messages received by a server (a technique previously implemented in TAO [45]). Conversely, Flick implements optimizations that IAC does not. In their publication about IAC, for example, Zalila et al. stated that statically calculating the size of the message buffer was ongoing work [104].

Although Zalila et al. criticize Flick's architecture, the structure of IAC is similar to Flick's in several respects. Most interestingly, IAC includes multiple implementations of the compilation step that translates an IDL-derived AST into constructs of Ada. These *tree converters* implement different presentations of the IDL-defined interface. Whereas

Flick uses different presentation generators to implement separate presentations (e.g., one for CORBA and another for ONC RPC), IAC uses them to implement parts of a single presentation—in particular, different output Ada files. This reflects a difference in the intended scopes of IAC and Flick. IAC was designed to be a "flexible and easily maintainable" CORBA IDL compiler for Ada [104, page 24]; Flick on the other hand was designed to support a broader range of input IDLs, presentation styles, and message transport systems for C and C++. This difference is also seen in the intermediate representations used by the two systems. Whereas IAC is driven by ASTs for CORBA IDL and Ada, Flick uses not just AST-like structures but also other IRs (PRES_C and MINT) to communicate information flexibility between compiler stages.

With respect to this dissertation, PolyORB and the IAC compiler are notable because they demonstrate ongoing interest in IDL compilers that, like Flick, combine flexibility and optimization. Flick apparently remains unique, however, as a "kit" for building IDL compilers that address flexibility and optimization goals at the same time.

5.1.3 Optimizing IDL Compilation for L4

Alongside its back ends for general-purpose middleware standards, Flick included back ends to produce stubs and skeletons for two operating systems, Mach [4] and Fluke [37]. These back ends were designed to be used the construction of microkernel-based operating systems—i.e., to implement efficient, RPC-based communication between the components of a microkernel architecture. Such optimized RPC has long been a concern of microkernel implementers. Bershad et al. described a lightweight remote procedure call implementation for Taos operating system more than twenty years ago [13], and since then, many microkernel efforts have included an IDL compiler to implement high-performance RPC atop the microkernel's basic IPC mechanisms. Mach included MIG, the Mach Interface Generator [73]; Fluke relied on Flick; Coyotos included the CapIDL compiler [89]; the in-development Barrelfish operating system includes the Flounder compiler [12]; and uniquely, the L4 microkernel family has utilized a series of different IDL compilers over time. The design of IDL compilers for L4 has been a recurring research topic, one driven by the desire for a compiler that is both flexible and extremely optimizing. Flick directly influenced this work.

Indeed, Volkmar Uhlig created one of the first IDL compilers for L4 by extending Flick: he implemented a new presentation generator and a new back end to create stubs and skeletons for use in multi-server operating systems built atop an L4 microkernel [97]. The extension demonstrated that Flick could be extended by people other than the primary Flick developers. However, Uhlig and his colleagues believed that their work was only partially successful. The problem was the Flick-generated stubs were too slow.

They used an I/O microbenchmark to measure the performance of their multi-server OS implementation, called SawMill Linux, against the performance of ordinary Linux. They found that the measured I/O operation required approximately 500 more cycles in SawMill Linux than it did in normal Linux—but they were expecting an overhead of only 200 cycles. Geffault et al. explained the cause of the unexpected overhead [43, page 114]:

> To our surprise, an important part of the problem seems to be the generated stub code. We expected cheap stubs since the Flick IDL compiler generates C code such that all its operations are inline generated in the user program. However, the code effectively generated by gcc [the C compiler] uses about 150 machine instructions for any client stub, mostly useless copies to and from local variables.... [It] is clear that an improved code generation facility has to be developed that generates near-optimal code which we found to be about 30 instructions.

In a subsequent publication by Haeberlen et al. [52, page 31], the SawMill project researchers further described the inefficiency of Flick-created stubs for SawMill Linux:

> [When] using the Flick IDL compiler for the SawMill Linux file system, we found that the generated user-level stub code consumed about 260 instructions per read request. When reading a 4K block from the file system, the stub code adds an overhead of about 17% due to stub instructions. (The stub code may also [generate] further indirect costs through side effects such as cache pollution.) For an industrial system, such overheads can no longer be ignored.

Uhlig and his colleagues did not formally characterize the causes of the run-time overhead in the Flick-generated stubs they produced. It is not clear, for example, how much overhead might be attributable to missing features in the L4 back end (a back end that Eide et al. did not implement), attributable to shortcomings in Flick's analyses, or due to missed code-optimization opportunities by the C compiler. All of these are likely contributors. Aigner wrote, for example, that stubs produced by Flick's L4 back end make a run-time choice between "long IPC" and "short IPC," which results in unoptimized code for the short-IPC case [5, page 18]. The stubs presumably make this choice even when Flick's message-size analysis could have be used to distinguish between the two IPC mechanisms at compile time. With respect to more general code optimization, as explained in Section 2.3, Flick primarily implements domain-specific optimizations and expects the target-language compiler to implement more general code optimizations. Haeberlen et al.

suggested that this was an ineffective strategy in practice [52] because the code output by Flick was too complicated.

What *is* clear from the quotes above is that run-time performance is a paramount concern for many operating-system developers. Overhead is measured not in milliseconds or microseconds, but in machine cycles and individual instructions. For an L4 IDL compiler, meeting such demanding performance requirements requires detailed and specialized knowledge of the target L4 environment, C compiler, and hardware platform. The developers of L4-based platforms felt that adding this knowledge to Flick would be difficult: Aigner estimated that the effort required to appropriately extend Flick for L4 would be similar to the effort required to implement a new IDL compiler [7, page 41].

The developers of L4-based systems therefore implemented a new generation of optimizing IDL compilers, ones designed specifically for L4. Andreas Haeberlen developed the IDL^4 compiler at the University of Karlsruhe [63]. Ronald Aigner at the Dresden University of Technology implemented DICE, the DROPS IDL Compiler [8], and Nicholas FitzRoy-Dale at the University of New South Wales created Magpie [29].

Haeberlen's IDL^4 compiler was specifically designed to "exploit specific knowledge of the microkernel, the hardware architecture, and the compiler that are being used, as well as certain characteristics of the application code" [51, page 4] in order to produce highly optimized RPC code for several versions of the L4 microkernel. Different versions of L4 support different features for IPC; with its built-in knowledge of specific kernel versions, the IDL^4 compiler can tailor its output to make the best use of any supported kernel. An example of architecture-specific optimization in IDL^4 is its ability to inline assembly-instruction sequences (x86) into the C code that it produces: it does this to generate stubs that are more efficient than those that would be produced by the C compiler in use. The IDL^4 compiler uses knowledge of the target C compiler in other ways as well. For example, IDL^4 outputs compiler-specific annotations into C code (e.g., to change the function-calling convention) and avoids constructs that IDL^4 knows lead to optimization problems in certain compilers. IDL^4 also implements an especially clever compiler-specific optimization called *direct stack transfer* [51, 52]. This optimization avoids a message-marshaling step by copying the client stub function's activation record (on the client's stack) to the server's stack; the server then uses the copied activation record to invoke its RPC service routine. The implementation of this optimization requires IDL^4 to generate assembly code, not only because it must copy a stack frame but also because it must specially lay out the stack frame to support features such as out parameters and register-based data transfer for small messages. Finally, IDL^4 supports IDL-level attributes that allow it to specialize generated code for particular uses. It provides attributes for specifying the typical and maximum sizes of variable-size data, for example, as well as for specifying caching policies for *flexpages*, which in L4 describe regions of virtual memory.

Haeberlen [51] compared IDL^4-generated stubs against those produced by the L4 version of Flick described previously. Over a set of microbenchmarks, he found that Flick-generated stubs contained between two and seven times more instructions than the equivalent stubs generated by IDL^4. The Flick-generated stubs were always slower, too. Haeberlen reported [51, page 36]: "The IDL^4 stubs we tested were 1.5–3 times faster than the corresponding Flick stubs; for domain-local servers, we even observed speedups of an order of magnitude. . . . [T]his improvement can only be explained by the special platform-specific optimizations we applied in IDL^4." For an I/O benchmark atop SawMill Linux, Haeberlen concluded that in comparison to Flick-generated stubs, the use of IDL^4-generated stubs improved the benchmark's throughput by 13% [51, 52].

In comparison to Flick, IDL^4 trades off certain kinds of flexibility in order to produce stubs that have high run-time performance. It incorporates specialized knowledge of the L4 toolchain—knowledge that Flick lacks—and thus can produce highly optimized code for L4. This specialization also makes IDL^4 rigid, however, and sensitive to changes in the toolchain. Haeberlen et al. suggested that generalizing the techniques use by IDL^4 would be an "obvious next step" and than incorporating those techniques into Flick would be "an ideal solution" [52, page 38].

Concurrent with Haeberlen's work on IDL^4, Ronald Aigner implemented DICE [6, 7], another IDL compiler for the L4 microkernel family. DICE and IDL^4 are similar in many respects: they both process interface descriptions written in CORBA IDL and DCE IDL, and they both produce stubs and skeletons with high run-time performance. They also use detailed knowledge of L4 and several similar techniques produce fast code. For example, both can inline (x86) assembly instructions into their generated stubs, and both can marshal a stub's parameter values in an order that differs from the order in which they appear in the stub function's signature. (Both compilers move fixed-size parameters to the front of the message buffer.) DICE does not, however, implement the direct stack transfer optimization that IDL^4 implements.

Although DICE and IDL^4 are similar, they are separate compilers with separate strengths, and their primary differences lie in two areas. The first is the particular versions of L4 that each best supports. The second area is flexible presentation: DICE supports annotations and stub styles that help users fit IDL-derived stubs into their existing programs.

Aigner compared stubs produced by DICE and IDL^4 in terms of their run-time performance on two versions of L4 [7]. Using a set of performance benchmarks, he identified ten RPC functions that were most commonly invoked in the benchmarks—and thus most important to their overall performance. On the "Fiasco" version of L4, the stubs produced by DICE were significantly faster than those produced by IDL^4: for most of the RPC functions, the average round-trip time of the DICE-produced stubs was less than half that of the IDL^4-produced stubs. Aigner also included Flick-derived stubs in his experiment. He found that they significantly outperformed IDL^4 stubs, but that they were mostly (but not always) outperformed by DICE stubs. He explained the results of his experiment [7, page 75]:

> Analysis of this data reveals that IDL^4 generated code is outperformed by all other stubs. This can be due to the fact, that IDL4 originally targeted L4 version X.0 [the "Hazelnut" version of L4]. The support for L4 version 2 ["Fiasco"] and L4 version X.2 (or version 4) has been added later and not much effort was spent on generating fast stubs. DICE generated code is faster than Flick generated code for most functions and especially for register IPC. For these functions its full optimization potential is recognizable.

On the "Hazelnut" version of L4, which is well supported by IDL^4, the qualitative results were significantly different. When Aigner ran his RPC benchmark on Hazelnut, he found that IDL^4 stubs were generally faster than those produced by DICE. He explained that this was due to the direct stack transfer optimization implemented by IDL^4 for Hazelnut [7, page 77]: "The gain due to optimization is about 300 cycles, which is 7%. This shows that architecture specific optimizations have a noticeable effect on performance." A more general lesson from Aigner's experiments might be that platform-specific optimizations in IDL compilers are brittle with respect to changes in the underlying platform. This conclusion is suggested not only by the differing relative performance of DICE- and IDL^4-produced stubs across different versions of L4, but also by the differing relative performance of IDL^4- and Flick-produced stubs in Aigner's and Haeberlen's work [7, 51]. The experiments performed by Aigner and Haeberlen are not directly comparable, but their differing qualitative results for IDL^4 and Flick are worthy of future study.

A second difference between IDL^4 and DICE is that includes several features that allow users to tailor IDL-derived stubs to their needs. For example, it allows users to direct the compiler to use predefined data types—e.g., those already defined by a client or server program—when it translates IDL definitions into C-language stubs and skeletons.[5] When DICE-produced code must marshal or unmarshal an instance of a user-defined type, it invokes a user-defined function to convert between the user-defined type and its on-the-wire message representation (which the user must also specify). DICE also supports message-passing stubs and a "decomposed" stub presentation style, similar to the one implemented by Eide et al. for Flick [24, 25], in which message marshaling and unmarshaling are separated from message transmission and receipt. Finally, it has a number of convenience features including the ability to insert tracing code into stubs and the ability to produce appropriate main loops for servers. Aigner states that DICE uses the *Abstract Factory* design pattern [39] to support multiple code-generation back ends. This is similar to Flick's support for multiple back ends (although one might say that Flick's back-end extension mechanism is more akin to *Strategy* than *Abstract Factory*).

Flexibility and extensibility were major motivators of a third IDL compiler for L4: the Magpie compiler [29], implemented by FitzRoy-Dale. In contrast to Flick, IDL^4, and DICE, which all construct their output using AST representations of C code, Magpie was driven by code-generation templates.[6] This strategy was chosen to address kernel-specificity problems as described above: e.g., that DICE and IDL^4 support a limited set of kernels, and their optimizations are brittle with respect to changes in L4.

FitzRoy-Dale described the problems that motivated his work [30, page 43]:

> . . . interface compilers for microkernels remain remarkably difficult to adapt to changing interface requirements. There are three major reasons for the difficulty: firstly, they tend to remain tightly-coupled to a small selection of kernels; secondly, the core interface-generation routines tend to be difficult to modify; and, finally, representation of target code is primitive compared with that offered by source-to-binary compilers.

Magpie's template system was intended to make it easy for L4 developers to maintain the IDL compiler—e.g., to keep Magpie up to date with respect to continual changes in L4.

[5]The IDL^4 compiler can import type declarations from a C++ header file, which it translates into IDL. This is different from DICE's support for user-defined types. DICE supports user-defined types through IDL annotations on individual operation parameters and through user-defined type-conversion functions.

[6]FitzRoy-Dale [30] references the template-based code-generation system that was added to Flick by Tim Stack [35], and notes that it is not used by most of Flick's back ends.

Moreover, the template system was intended to facilitate modifications by developers who were not familiar with the internals of the Magpie compiler. FitzRoy-Dale reported that the template system was only partially successful in meeting these goals [30, page 45]: "Perhaps the best proof of both the success and failure of Magpie is the fact that after more than a year of use, many different templates were successfully developed to accommodate changing requirements, but the sole developer of new templates was the implementor of Magpie [FitzRoy-Dale himself]."

Template-driven code generation can make it difficult for an IDL compiler to perform optimizations on the code it generates, as described in Section 5.1.1. Magpie does implement some important optimizations—notably, like DICE and IDL^4, Magpie supports multiple back ends for multiple target platforms, and it can output assembly instructions into the C code that it generates. However, little to nothing has been published about the run-time performance of Magpie-generated stubs to date. Aigner, for example, only compared DICE to Magpie in terms of *the amount of code in the compilers themselves*—not the output stubs' performance [7]. Rather than focusing on optimization with the context of a template-driven compiler, FitzRoy-Dale suggested replacing templates with a transformation system based on logic programming [30]. (He characterized Flick's approach—the use of multiple intermediate representations for separating concerns—as "inflexible" [30, page 46].) The new IDL compiler he proposed, Currawong, was not implemented. Currawong subsequently became the name of FitzRoy-Dale's tool for "system software architecture optimization" [31, 32].

Flick was often cited by the creators of IDL^4, DICE, and Magpie for its flexibility and optimization features; in addition, Uhlig's extension of Flick for L4 was used as a benchmark for the evaluation of performance-optimized RPC code. Work in the area of optimized RPC for microkernels is ongoing. This is exemplified by the Flounder IDL compiler [12] for Barrelfish, a current operating system project at ETH Zurich. For the L4 microkernel family, recent work has included a proposal to obtain flexible and optimized RPC code by abandoning IDL compilers altogether. Feske developed a *dynamic RPC* system in which microkernel programmers write stub code by hand using C++ stream operators, as illustrated in the following example [27, page 42]:

```
Ipc_client client(dst, &snd_buf, &rcv_buf);
int result;
client << OPCODE_FUNC1 << 1 << 2
       << IPC_CALL >> result;
```

Feske argues that such code is easy for kernel developers to write, understand, and maintain. He claims that in contrast the effort required to develop, validate, and maintain an IDL compiler, the effort demanded by the C++ stream library is minimal. He points out that although hand-written marshaling code admits the possibility of message-format mismatches between clients and servers, such errors can be minimized by encapsulating actual communication code in RPC stubs (as IDL compilers do) and then sharing the stubs' function signatures across client and server programs.[7]

Feske presents results to show that the run-time performance of dynamic RPC stubs can be better than the performance of equivalent DICE-generated stubs. He attributes this result to the relative simplicity of dynamic RPC stubs and the ability of modern C++ compilers to perform sophisticated optimizations [27, page 45]: "The relatively rigid code as generated by DICE leaves less potential for automated compiler optimization than the high-level C++ stream code." Aigner compared a similar experiment, comparing the run-time performance of DICE-generated stubs and dynamic RPC stubs, and reached a qualitatively different conclusion [7, page 75]. In Aigner's experiment, DICE-generated stubs consistently (but not always) outperformed dynamic RPC stubs.

Although the reported performance results are inconclusive, they suggest that the complexity of IDL compilers may be unwarranted for microkernel development. As Bershad et al. observed [13], the RPC interfaces in microkernels are generally simple; the primary optimization problem lies not in handling complex data types but in fully utilizing the microkernel's features for IPC. A stream abstraction for RPC, combined with the optimizations implemented by modern C++ compilers, may be sufficient for addressing such issues. Flick's optimizations generally operated at a higher level of abstraction: their intent was to perform domain-specific optimizations that were outside the scope of the target language compiler. This included analyses over whole messages and complex data types, which are used in RPC-based systems in general but perhaps not commonly within microkernels.

On the other hand, even if Flick's complex optimizations are not often required, this is not to say that IDL compilers for microkernels have no optimizations to offer. For example, it is not apparent how the direct stack transfer optimization, implemented by IDL[4], would be easily achieved in Feske's dynamic RPC system. The development of IDL compilers for

[7]Feske's dynamic RPC work is thus similar in spirit Gokhale and Schmidt's work in optimizing the stream-based IIOP layer in TAO [46, 47], which was discussed in Section 5.1.2.

operating system development is thus not yet over: in fact, Aigner's [7] and Feske's [27] recently published results suggest that future research is warranted.

5.2 Novel Realizations of Design Patterns

Chapter 3 presents a novel technique for the realization of design patterns in software systems, a technique that separates the static parts of a pattern from the dynamic parts. The static pattern elements are realized as components or component interconnections that are fixed at compile time, whereas the dynamic elements are realized as objects or object references at run time. The static elements are specified in a component-definition and linking language that is separate from the programming language used to implement the software system's parts. By making static elements explicit and obvious, this "two level" pattern-realization approach can aid static analyses, enable the checking of design rules, and promote compile-time optimizations that improve run-time performance. Chapter 3 demonstrates the approach using Knit [82], a component language for C, and the OSKit [36], a set of components for building operating systems.

The work presented in Chapter 3 thus deals with three topics that are unusual in the design-pattern literature. The first is the idea that a software developer might explicitly choose between static and dynamic mechanisms for pattern elements, as opposed to always using the mechanism prescribed by the conventional, object-oriented realization of a pattern. The second is the application domain: i.e., using design patterns within component-based operating systems. The third is the more general idea of implementing object-oriented design patterns in C, which is not an object-oriented language. The following sections describe work in these three areas.

5.2.1 Controlling Static and Dynamic Structure

The method described in Section 3.3.1 requires that a software architect (1) identify the parts of a design pattern that correspond to static information about the software system being developed, (2) "lift" that knowledge out of the implementations of the system's components, and then (3) encode that knowledge within component definitions and connections, which are processed at compile time. This means that at the time the pattern is applied—i.e., when the system is designed—each pattern element is classified as either static or dynamic. Subsequent research has addressed the idea that, in some situations, it is advantageous to defer the choice between static and dynamic realization of software features. This deferment is enabled by new variability mechanisms that allow decisions about static and dynamic binding to be postponed until compile time. In modern software platforms that support just-in-time compilation, compile time may be very late indeed in the software deployment process.

Chakravarthy, Regehr, and Eide proposed *edicts*, a variability mechanism that makes it possible for a developer to choose between compile-time and run-time bindings for features within a software system. As described by the authors [14, page 109], the edict approach “changes the binding time of a feature from being a design-time attribute of a product line to being an assembly-time attribute of a product.” In other words, the person who configures a particular software system from parts can decide, for each edict-based connection, whether the connection should be implemented in a static fashion or a dynamic fashion. The binding sites for an edict—i.e., the realization of the edict in code—is implemented using a combination of design patterns and aspect-oriented programming; Chakravarthy et al. reference the published version of Chapter 3 for describing how design patterns may be used to implement decisions that are set at compile time.

The binding-time flexibility provided by edicts is useful for creating software product lines in which different products have significantly different requirements for feature binding. Chakravarthy and his colleagues demonstrated this flexibility by creating a product line based on JacORB, an implementation of CORBA with many configurable features. With edicts configured to support dynamic binding, one can create a full-featured middleware platform appropriate for resource-rich computing environments. With edicts configured to use static binding, Chakravarthy created a middleware platform more suited to a resource-constrained environment, such as a smart phone. Static binding made it possible for a compile-time analyzer—a tool called ProGuard [64]—to more effectively optimize JacORB for an embedded environment and for a particular application.

As the authors explained [14, pages 115–116], “The use of design patterns and edicts does not produce static optimizations by itself. Rather, as we show, our technique allows static code optimizers to be much more effective.” They performed an experiment in which they used ProGuard to optimize JacORB for use by a particular application: they optimized both the original version of JacORB and the version of JacORB with edicts. The application’s code explicitly selected particular JacORB features: in principle, a program optimizer could use this information to optimize the original JacORB implementation, in which features are selected dynamically. The experiment showed, however, that ProGuard was unable to perform the required analysis. After optimization by ProGuard, the static code size of the edict-enabled JacORB was 32.2% less than that of JacORB without edicts. Based on their results, Chakravarthy et al. concluded that “edicts help to ensure that opportunities for static optimization are not lost” [14, page 119]. The design-pattern approach described in Chapter 3 provides a similar potential benefit. Edicts, however, provide the additional benefit of binding-time flexibility.

Subsequent to Chakravarthy’s publication on edicts, Rosenmüller et al. presented a different variability mechanism for implementing features that require flexible binding times [84, 85]. Their approach was based on FeatureC++ [10], an extension of the C++ language for feature-oriented programming [81]. A *feature* in FeatureC++ contains a set of C++ definitions as well as *refinements* that extend definitions from other features; a refinement is a mixin that may introduce members to an existing class and extend existing methods. A complete software product is represented as a stack of features, in the style of mixin layers [92]. The bottom-most feature provides the base implementation of the software, and every other feature in the stack implements an extension to the composition of features below it.

Rosenmüller and his colleagues extended the FeatureC++ compiler to support both the static and dynamic composition of features [85]. When a software developer requests static binding for a feature, the compiler combines the code of the feature with the code of the features that it extends; the composed feature is then compiled as a single unit. Such combinations of statically combined features are called *dynamic binding units* [84]. When a developer requests *dynamic* binding for a feature, the compiler generates code that will invoke the feature at run time, if the feature is actually selected at run time. Every class refinement in the dynamically bound feature becomes a decorator of the base class (in the style of the *Decorator* design pattern [39]), and the FeatureC++ compiler transforms the base class so that it invokes decorators at the appropriate times. The FeatureC++ compiler supports binding-time choice on a per-feature basis.

Binding-time flexibility in FeatureC++ is thus similar to edicts: in each approach, a single variability mechanism is designed to support both static and dynamic binding. As Rosenmüller et al. point out, the edict and FeatureC++ approaches differ in that edicts are based on hand-implemented patterns and aspects, whereas FeatureC++ implements binding-time variation through different styles of compilation and code generation [85]. FeatureC++ also supports validation of configurations against feature models, which edicts do not inherently support [84].

Several modern languages for aspect-oriented programming, including AspectJ [60], support aspect weaving at different points in the software-deployment process. Commonly, they support the incorporation of aspects when the "base code" of the system is compiled, and they support the incorporation of aspects when the base code is loaded just prior to execution. This provides a style of binding-time flexibility—but one that is often not exploited for improving the run-time performance of programs, which is the focus of this dissertation. Support for multiple aspect weaving times tends to focus on *who* is able to deploy aspects within a program—the original developers or "downstream" consumers—not on the effects enabled by differing weaving times, such as static optimization for performance and code size.

5.2.2 Realizing Design Patterns in Component-Based Operating Systems

Chapter 3 demonstrates the novel realization of design patterns with two examples based on the OSKit [36], a large collection of operating system components. OSKit components are defined using Knit [82], a component definition and linking language for systems software, and the functions within components are written in C. The design of component-based operating systems has been a popular and recurring topic in the systems research community; notable implementations include the OSKit, MMLite [54], Pebble [38], Think [26], PECOS [44], BOTS [75], and CAmkES [62]. Knit and the OSKit are commonly cited as related work in this field. However, it is *uncommon* for the designers of component-based operating systems to describe the design or use of their component collections in terms of design patterns.

The designers of TinyOS are a notable exception to this rule. TinyOS [65] is a component-based operating system for wireless sensor network devices; its components are written in nesC [42], a dialect of C with extensions for component-based programming in which component assemblies are fixed at compile time (as in Knit). Gay et al. present a set of eight design patterns that are commonly used in the implementation of TinyOS-based systems [41]. The patterns descriptions follow the "Gang of Four" style developed by Gamma et al. [39], and in fact, three of the eight patterns are based directly on the object-oriented patterns presented by Gamma et al.: *Adapter*, *Decorator*, and *Facade*. Gay and his colleagues describe the structures of TinyOS patterns using the facilities of the nesC programming language. In particular, nesC components represent statically defined participants, and nesC component interconnections represent statically defined relationships. This kind of pattern structure is similar to the realization of object-oriented patterns described in Chapter 3. This is seen most clearly in the TinyOS versions of *Adapter*, *Decorator*, and *Facade*: the TinyOS pattern structures are essentially identical to the results of applying the method of Chapter 3 to the respective object-oriented patterns, and then realizing the patterns with Knit. The intent of Gay et al. was to present a catalog of patterns useful for TinyOS programming, so they do not discuss a general technique for expressing object-oriented patterns at the level of components (unlike Chapter 3). Although some of the patterns in the TinyOS catalog are essentially the same as the Knit-based realizations of the corresponding object-oriented patterns, Gay et al. do not cite Knit or the published version of Chapter 3 as related work.

SNACK [48] by Greenstein et al. is another component-based operating system for wireless sensor network nodes. Like TinyOS, SNACK components are implemented in nesC. Unlike TinyOS, SNACK-based applications are defined through a configuration language that directly supports certain types of compositions based on common design patterns. The SNACK composition language provides a *transitive arrow connector*, for example, which allows *Decorator*-like interposition on inter-component connections. Greenstein and his colleagues describe the realization of other design patterns in SNACK, which they characterize as "program methodologies that can improve the behavior of sensor applications" [48, page 73]. These are based on uses of the transitive arrow connector and *services*, which are combinations of primitive components. In comparison to Knit, SNACK compositions are expressed more abstractly: SNACK's abstractions allow a programmer to directly express certain composition styles, such as decoration, that require encoding using Knit's notion of explicitly defined component links. The method described in Chapter 3 can be thus been seen as a recipe for encoding relationships that, in some cases, SNACK can realize more straightforwardly. Greenstein et al. cite Knit and the OSKit as related work, but do not reference the published version of the work presented in Chapter 3.

A third notable system is Koala [99, 100], implemented by van Ommering et al.: Koala is a component model and compiler tailored to the production of embedded software for consumer electronics. Its notions of component definition and linking are similar to those in Knit. As in Knit, the software within a Koala component is written in C. Also like Knit, Koala provides a domain-specific language for defining component types and defining assemblies of component instances.

Unlike Knit's component-definition language, however, Koala's language provides built-in support for product-line diversity through *diversity interfaces* and a design pattern called a *switch* [100]. A diversity interface imports system-configuration values into a component: e.g., compile-time constants that represent the selection or deselection of crosscutting features. This information can be used to generate efficient code when an assembly of components is compiled. A switch is a component that selects between two or more components that provide different implementations of the same functionality—for example, two components that implement control functions for two different pieces of hardware. The switch's job is to direct service requests (i.e., function calls) from a client component to the appropriate service implementation. A switch also has a control interface that directs the flow of requests through the switch: this is commonly a diversity interface, which allows the switching decision to be made at compile time. When a switch decision is set at compile time, Koala is able to perform optimizations such as automatically removing unreachable components.[8]

Koala demonstrates the value of realizing certain design patterns at the level of components and utilizing those patterns to obtain performance benefits at run time—the benefits that are also the goal of the pattern technique presented in Chapter 3. By supporting certain patterns in its configuration language, Koala helps to ensure that those patterns do not become lost among all the component interconnections that are part of a complete application—a potential problem with the technique described in Chapter 3. On the other hand, unlike the presented technique, Koala does not address the idea of realizing object-oriented patterns at the level of component compositions in general.

The Cake language by Stephen Kell also seeks to realize particular design patterns at the level of components: in particular, the Cake compiler generates adapters for components written in C [58]. Cake is a declarative, rule-based language for describing how two or more component interfaces relate. A Cake programmer declares a set of *correspondences* between the elements of two or more interfaces; from this specification, which can be quite sophisticated, Cake creates the code for an adapter that allows components with the described interfaces to interact. Knit and Koala both had limited inherent support for interface adaptation through symbol renaming; Cake's support for adaptation is obviously

[8]Koala switches are thus related to Chakravarthy's variability mechanism based on edicts [14], discussed in Section 5.2.1. A Koala switch modularizes a feature choice, is set by a control interface, and supports static optimization when the switch is set at compile time. In Chakravarthy's work, a design pattern modularizes a feature choice, and the choice is controlled by an aspect (an edict). When the choice is set at compile time, the pattern and edict promote compile-time optimization.

much more significant. Chapter 3 describes the correspondence between the object-oriented *Adapter* pattern and the design of adapter components, but the method presented there does not relieve a Knit programmer from the task of actually implementing adaptation logic. Cake addresses the latter problem. Kell writes that Cake was influenced by Knit, but that Cake "radically extends [Knit's] adaptation capabilities" [58, page 339]. Kell cites the published version of Chapter 3 for arguing that component languages like Knit's are sometimes superior for expressing software composition [57, page 9]: "their simplicity admits more automated reasoning and doesn't introduced unnecessary dynamism."

In contrast to the component systems described above, which provide special support for only a few component-composition design patterns, the OpenCom model by Coulson et al. [16] seeks to be more general. As described by Coulson and his colleagues, OpenCom was designed to be a general-purpose component model for systems programming [16, page 1:3]:

> OpenCom tries to maximize the genericity and abstraction potential of the component based programming model while at the same time supporting a principled approach to supporting the unique requirements of a wide range of target domains and deployment environments. This is achieved by splitting the programming model into a simple, efficient, minimal kernel, and then providing on top of this a principled set of extension mechanisms that allow the necessary tailoring.

The OpenCom kernel defines a minimal component model that supports dynamic component loading, linking, and unloading. Atop this kernel, OpenCom allows a programmer to define more specialized *component frameworks*. A component framework does three things: it provides a set of components to address a focused area of concern, such as operating-system programming; it defines a model for extending those components; and it provides constraints on the ways in which components may be extended. A component framework thus defines an environment for component-based programming and has significant control over programs within that environment—essentially, it defines an architectural style [40]. A component framework uses reflective metamodels, provided by OpenCom, to examine and control the component-based programs defined within the framework. Coulson and his colleagues report that OpenCom has been used to implement several types of systems software including middleware, embedded systems, a programmable networking [16].

The ability to define and enforce an architectural style is a powerful system-building tool, and one can imagine this power being applied to styles based on component-based realizations of design patterns. Section 3.4.1 describes some of the analyses that Knit provides that are useful for reasoning about the use of patterns within component assemblies. One expects that OpenCom's frameworks could provide similar and additional support for reasoning about component-level patterns.

It is, however, apparent that the overall goals of OpenCom and Knit are somewhat different. The method described in Chapter 3 identifies the *static participants* within a pattern instance and realizes them so as to promote compile-time code optimization. In comparing OpenCom to Knit, however, Coulson et al. write that Knit's main shortcoming is its compile-time component model [16, page 1:38]: "The main limitation of Knit is that it addresses purely build-time concerns: the component model is not visible at runtime, so there is no systematic support for dynamic component loading, still less managed reconfiguration." Such dynamism is inherent in OpenCom's model of components. Even if OpenCom might support reasoning over the use of design patterns among components, its dynamic component model would hinder compile-time optimizations for performance—optimizations that are the goal of the novel realizations of variability mechanisms presented in this dissertation.

5.2.3 Realizing Object-Oriented Design Patterns in C

Chapter 3 shows how object-oriented design patterns can be realized in a dialect of C with components, with particular focus on "lifting" static pattern participants to the level of components. A few publications have described the realization of object-oriented design patterns in plain C, as summarized below. In comparison to the vast literature on implementing design patterns in object-oriented languages, however, published work on implementing design patterns in C is quite limited.

To realize any design pattern within code, one must decide how the pattern participants will be implemented. More particularly, when the C implementation of a pattern involves run-time "objects," a programmer must explicitly decide how to represent those objects in C, because C is not an object-oriented language. A typical programming solution uses `struct` types to represent classes: instances of the structure types represent objects at run time, and function pointers within the structure instances encode dynamically dispatched methods. This is not, however, the only possible encoding of classes and objects.

In his book about C-based design patterns for embedded systems [23], for example, Douglass describes three different techniques for implementing classes. The first "is simply to use the file as the encapsulating boundary; public variables and functions can be made visible in the header file, while the implementation file holds the function bodies and private variables and functions" [23, page 10]. This approach is the plain C analogue of using Knit component instances as pattern participants, one of the key ideas presented in Chapter 3. Douglass' second approach to implementing classes is through `struct` types: a class is described by a header file that declares both (1) a structure type and (2) a set of functions, where each function corresponds to a public method of the class. The functions are defined in a corresponding implementation file, along with any functions that represent private methods. Douglass' third method extends the `struct` types with function-pointer members—the typical programming solution already described—in order to support dynamic method dispatch. The presentation of implementation *choices* in Douglass' work, as opposed to the presentation a single implementation *recipe*, supports the general claim of Chapter 3: that it is useful to choose the realization of pattern participants, based on the context of the pattern's use, in order to obtain benefits such as performance optimization.

Prior to the publication of Douglass' book, Petersen published a series of articles about implementing well-known object-oriented design patterns in C. The first article in the series presents a pattern for realizing classes: Petersen calls this the *First-Class ADT* pattern [76], and it is the same as the second implementation technique presented by Douglass. Using *First-Class ADT*, Petersen's subsequent articles discuss C implementations of the *State* [77], *Strategy* [78], *Observer* [79], and *Reactor* [80] patterns. These are largely straightforward, and in each, pattern participants correspond to data-structure instances that are created at run time. Unlike Douglass or the work presented in Chapter 3, Petersen does not explore the idea that a pattern's participants might be realized in different ways across different uses of the pattern.

Although publications about design patterns in C are somewhat rare, people in the design patterns community continue to stress that patterns are meant to be neither language-specific nor rote. As Jason Smith wrote in his 2012 book about deconstructing design patterns [93, page 11]:

> Patterns are language-independent concepts; they take form and become concrete solutions only when you implement them within a particular language with a given set of language features and constructs.

Smith also points out that the application of a pattern is intended to be an activity in which a programmer makes conscious decisions [93, page 9]:

> Patterns are intended to be mutated, to be warped and molded, to meet the needs of the particular forces at play in the context of the problem, but all too often, a developer simply copies and pastes the sample code from a patterns text or site and declares the result a successful application of the pattern. This is usually a recipe for failure instead of a recipe for producing a good design.

The pattern-realization technique presented in Chapter 3 is based on both of these points: patterns as language-independent notions, and patterns as concepts that can be realized in myriad ways. These points are also evident in Smith's graphical notation for describing patterns, called *Pattern Instance Notation (PIN)*, which avoids the object-oriented constructs of UML and instead describes patterns in terms of roles and hierarchy. The pattern-realization component diagrams in Chapter 3 resemble PIN drawings to some extent. Although this similarity is almost certainly due to independent invention—Smith cites neither the published version of Chapter 3 nor previous work about the *unit* model of components [33, 34]—it is a measure of evidence that the work presented in Chapter 3 addresses concerns that continue to be relevant to the software-engineering community.

5.3 Dynamic CPU Management

The CPU Broker, described in Chapter 4, is a novel variability mechanism for composing real-time tasks within a larger software system. It mediates CPU-allocation requests between a set of real-time tasks and the scheduling facilities of a real-time operating system (RTOS). Using run-time feedback from the tasks that it manages, the CPU Broker adjusts the tasks' CPU allocations to ensure that the overall software system maintains a high application-level quality of service (QoS), insofar as possible, even in the face of dynamic changes to available resources, the set of managed tasks, and the relative importances of tasks. Because the CPU Broker seeks to keep a system running even under adverse circumstances, it can be seen as a building block for self-adaptive systems [86], a field of study sometimes known as autonomic computing [59].

The CPU Broker has been regularly cited by researchers who build subsequent, related infrastructure for managing CPU resources in a coordinated fashion across a set of real-time tasks. The following sections summarize this related work and organize it into three areas. The first includes broker-like systems that explore design and implementation choices that differ from those made for the CPU Broker (Section 5.3.1). For example, whereas the CPU Broker is implemented in middleware, other brokers have been implemented within operating system kernels. The second area includes work toward improving the prediction of tasks' future CPU needs (Section 5.3.2). The CPU Broker architecture includes *advocates*, which make predictions on behalf of tasks, but leaves the details of accurate prediction largely unexplored. The third area covers adaptive systems that manage multiple kinds of resources simultaneously (Section 5.3.3). The CPU Broker was incorporated into one such system and was included in dry runs of the capstone technology demonstration for the DARPA-funded PCES program.

The literature summary below shows that resource brokering continues to be relevant to the design of self-adaptive systems. Complex systems benefit from stabilizing infrastructure, such as the CPU Broker, that acts at run time to optimize a system's utility. The ongoing series of publications that cite the CPU Broker indicate that the design, implementation, and evaluation of such infrastructure are still concerns for computer-systems research.

5.3.1 Broker-Like Systems for CPU Scheduling

The CPU Broker embodies a set of design and implementation decisions toward the goal of managing CPU resources for a set of real-time tasks. Subsequent systems for CPU management carried some of these elements forward while exploring different implementation strategies and addressing new concerns. As one may recall from Section 4.3 and Section 4.4, the CPU Broker's architecture includes *advocate* objects, which make CPU requests on behalf of tasks, and *policy* objects, which manage the requests of multiple tasks, resolve conflicts, and interact with the scheduling facility of an underlying RTOS. These objects can generally be imposed upon the managed tasks in a noninvasive manner: e.g., advocates can be inserted into middleware or implemented as separate processes that monitor tasks. In either case, the architecture helps to decouple tasks' "application logic" from the implementation of tasks' real-time behavior. This decoupling makes it possible for a system designer to define real-time policies relatively late in the software life cycle: not when individual tasks are implemented, but later, when an overall system is assembled or executed. The central parts of CPU Broker are implemented in a CORBA middleware-based server, which manages CPU reservations at run time. An important advantage of the middleware-based implementation is that it is relatively easy for programmers to configure and extend the CPU Broker with custom advocates and policies.

Several research efforts have explored broker-like systems that place their adaptation strategies not in middleware, but in an operating system kernel. Prior to the development of the CPU Broker, Shenoy et al. explored the trade-offs between middleware-based and OS-based support for scheduling multimedia applications [91]. Based on their experiments with QLinux, an enhanced version of Linux, and TAO, an implementation of CORBA, they concluded [91, page 32]:

> Our results showed that although the run-time overheads of a middleware can impact application performance, user-level resource management can, nevertheless, be just as effective as native OS mechanisms for certain applications. We also found that kernel-based mechanisms can be more effective at providing application isolation than a middleware system.

The CPU Broker represents a hybrid approach. It places the adaptive scheduling components—advocates and policies—at user level, but relies on a real-time operating system to actually perform CPU scheduling.

Abeni et al. developed a real-time scheduling system with similarities to the CPU Broker, but which places all of the scheduling components inside an operating system [2].[9] Like the CPU Broker, Abeni's scheduling system implements *adaptive reservations* [1]. Its organization is also similar to that of the CPU Broker. Each task is associated with a manager, and all task managers communicate with a central supervisor. A task manager plays a dual role, like a CPU Broker advocate, in that it collects performance data from its task and also issues predictions of future CPU requirements to the central supervisor. The supervisor in Abeni's system is akin to a CPU Broker policy, making global decisions about how reservation requests should be adjusted.

The CPU Broker and the system by Abeni et al. differ in terms of their implementation strategies, software architectures, and research focus. From an implementation perspective, the primary difference between the CPU Broker and the system implemented by Abeni et al. is that the former is implemented in user space and the latter is implemented within an operating system. Abeni and his colleagues refer to the CPU Broker as an "interesting proposal" at the middleware level and distinguish their work as implemented "for a general purpose operating system by a minimally invasive set of modifications" [2, page 133]. In terms of software architecture, the primary difference between the two systems lies in how the entities within each system compose. The CPU Broker allows one to assemble multiple advocates for a single task: by combining advocates in a *Decorator*-like fashion, a

[9] The journal article by Abeni et al. [2] is a revised version of an RTAS 2004 conference paper by the same authors [22]. Coincidentally, the conference paper about the CPU Broker, reprinted in Chapter 4, also appeared at RTAS 2004.

programmer can create new advocating strategies by combining simpler ones (Section 4.3.1). Similarly, the CPU Broker allows a programmer to compose multiple policies, which can be useful when tasks are grouped into multiple scheduling classes (Section 4.3.2). The system described by Abeni et al. supports multiple implementations different managers and supervisors—providing different prediction and control policies—but does not directly support the idea of implementing new ones through the composition of existing ones. Finally, in terms of research focus, the presentations of the two systems differ in terms of the concerns addressed. Each system is a realization of adaptive reservation-based scheduling. The work presented in Chapter 4, however, has a particular focus on openness and ease of extension by system designers. The research by Abeni et al., in contrast, has a focus on developing prediction and control strategies that are informed by control theory. This aspect of their work is discussed in Section 5.3.2, below.

Abeni's coauthors continued to evolve their Linux-based scheduling system and the result was *AQuoSA*, the Adaptive Quality of Service Architecture [74]. AQuoSA retains the general architecture already described, i.e., task-specific managers communicating with a central supervisor. The "minimally invasive" [2, page 133] implementation strategy is also carried forward. Most of AQuoSA is implemented as Linux kernel modules, which ultimately depend on a small patch to the standard Linux scheduler. Unlike the authors' previous Linux-based work [2, 22], however, AQuoSA includes user-space libraries that allow programmers to implement task managers and supervisors outside of the Linux kernel. The authors of AQuoSA, Palopoli et al., also retained their research focus on developing the theory of adaptive reservations: they present a model that "allows one to build a theoretically well-founded adaptive control law, split across a predictor and a controller, acting locally on each task, where control goals are formally specified, and conditions for their achievement are formally identified" [74, page 4].

Because AQuoSA is similar to the previous work of Abeni et al. [2], the two systems are similarly related to the CPU Broker and the work presented in Chapter 4. The previously presented summary of software architecture and research focus applies to a comparison of AQuoSA and the CPU Broker: the CPU Broker allows a programmer to create new advocates and policies by composing existing ones, which AQuoSA does not; AQuoSA provides adaptive scheduling components based on a mathematical control theory, which the CPU Broker does not. (Palopoli et al. state that, in published paper about the CPU Broker, "most of the work is on the architectural side" [74, page 4].) AQuoSA and the previous scheduler by Abeni et al. compare differently, however, to the CPU Broker in terms of their supported implementation strategies for adaptive scheduling components. The system by Abeni et al. supports adaptation components inside of a Linux kernel, and the CPU Broker supports adaptation components outside the kernel, at user level. AQuoSA supports components that reside at both levels, and in this respect, it improves on both the CPU Broker and the scheduler by Abeni and his colleagues.

In contrast to the CPU Broker's design, where adaptive scheduling components run at user level, Abeni, Palopoli, and their colleagues explored operating system support for feedback-driven, adaptive scheduling. Kalogeraki et al. explored a different point in the design of adaptive scheduling systems. Whereas the CPU Broker manages adaptation for tasks on a single device, Kalogeraki et al. implemented a resource manager that performs adaptive, feedback-driven scheduling for distributed—i.e., multiple-device—real-time object systems [56].

The architecture of Kalogeraki's resource-management system includes distributed *Profiler* and *Scheduler* entities—each node of the distributed system hosts one of each—that communicate with a central *Resource Manager*. As the names of these objects suggest, a Profiler collects data about the execution of tasks on a processor, a Resource Manager makes resource-allocation and task-migration decisions in response to collected data, and a Scheduler implements the current decisions by scheduling tasks on a processor. A Scheduler performs *least-laxity scheduling* of tasks: each task is modeled as a graph of method invocations on CORBA objects, and a Scheduler controls the middleware in order to schedule tasks' method invocations according to the estimated residual laxity of the competing tasks. (The estimated residual laxity of a task is the difference between the task's deadline and the estimated time of the task's completion.) As described by Kalogeraki and her colleagues, their resource-management system as a whole is driven by three feedback loops. The first and finest-grain loop measures elapsed time on a single node in order to update the estimated residual laxity values of tasks. The second loop, which runs much less frequently than the first, uses measured time values and other data to refine the initial estimates of the laxities for tasks. These initial estimates are based on past timings of method invocations on a node, the number of tasks currently running, and information about inter-task dependencies. The third feedback loop is driven by measured processor loads and task-laxity values collected across the distributed system: these values are used by the central Resource Manager to decide how new objects will be allocated to processors and how existing objects will be migrated between processors in the distributed system.

The architecture and implementation of Kalogeraki's resource-management system is similar to those of the CPU Broker in some respects, but in the traits that matter most, the two systems are quite different. Kalogeraki's system, like the CPU Broker, performs adaptive, feedback-driven scheduling for soft real-time tasks. Furthermore, both systems are designed to integrate with applications based on CORBA middleware, and both systems use middleware to implement the components that make adaptation decisions. The most notable difference between the two systems is that Kalogeraki's system manages scheduling for distributed systems, whereas the CPU Broker manages resources only for a single processor. (Kalogeraki and her colleagues state [56, page 1159]: "Like our system, the CPU Broker monitors resource usage and adjusts allocations in a non-intrusive manner, but it does not address distributed scheduling.") Two additional differences are also worth noting. The first is that Kalogeraki's resource-management system implements a scheduler, whereas the CPU Broker does not: the CPU Broker performs mediation among tasks, but relies on the facilities of an underlying operating system to actually perform scheduling. This architectural difference represents a trade-off: for example, Kalogeraki's system can make finer-grain scheduling decisions, but the CPU Broker can mediate tasks that are not middleware-based. The second notable difference lies in the two systems' support for extension by programmers. As presented by Kalogeraki and her colleagues, their resource-management system is largely a closed and complete system. In contrast, and as already described, a central feature of the CPU Broker is its support for user-custom scheduling advocates and policies.

The architecture of the CPU Broker was designed to help separate "application logic" from the implementation of real-time behavior, to modularize the implementation of inter-application policies, and to make it possible for system designers to deploy policies late in the software life cycle, when a system is assembled from separately developed tasks. In his Ph.D. dissertation, Andrew Wils describes a different software architecture that addresses these same concerns, but which takes a more holistic approach to the design of timing-driven, self-adaptive systems [103].

Wils' approach is component-based: an application is structured as an assembly of components that communicate via messages. Separate from the components that implement the application's purpose, *constraint monitors* observe the flow and timing of messages. Monitors report observations to *decision makers*, which determine when adaptation is required; the decision makers in turn invoke *adaptation actors*, which can change components' configurations and reroute messages between components. Wils' architecture is also hierarchical. Application instances are managed by *application timing managers*, which communicate with an *inter-application timing manager* that makes decisions and allocates resources across all the applications that are running on a device. Wils' dissertation describes how the timing managers communicate through *resource contracts* that describe applications' requirements and flexibility. Wils presents an implementation of his model in a system called CRuMB—"Component Runtime support for a Monitoring based Broker" [103, page 77]—and in addition, describes how his techniques for implementing timing-driven adaptivity influence the software development process.

The CPU Broker is analogous to the inter-application timing manager in the architecture summarized above. The interactions between Wils' timing managers are based on resource contracts [103, page 52], and Wils refers to the CPU Broker as an example of systems that negotiate resource contracts between tasks [103, page 20]. Wils states that resource contracts are "only part of the solution" [103, page 20] for building self-adaptive systems, however, and this opinion is evidenced by the many features in Wils' architectural approach that go beyond the capabilities and scope of the CPU Broker. Most obviously, the CPU Broker does not address how an application might be built to adapt its own behavior in response to resource shortages. Other researchers have used the CPU Broker in combination with other techniques for adapting applications' quality of service; that work is summarized in Section 5.3.3, below.

5.3.2 Improved Feedback-Driven Prediction

The architecture of the CPU Broker includes advocates, which predict the future CPU requirements of individual tasks, and policies, which decide how to allocate CPU resources among competing tasks. Section 4.4.3 presents an example set of general-purpose advocates and policies, and Section 4.5.3 describes the construction of an application-specific advocate. Beyond these examples, however, Chapter 4 does not provide details about how one can design and implement "good" advocates and policies. Good advocates are ones that make accurate predictions. Good policies lead to desirable system properties, such as fast reconfiguration in response to changing environmental conditions and stable configuration during of stable conditions.

Subsequent to the work presented in Chapter 4, researchers continued to investigate techniques for improved CPU prediction and response, and the rest of this section summarizes this work in two general categories. The first includes efforts to make better predictions through better data collection. The second includes research on adaptive, real-time systems that are based on control theory.

Toward the goal of better data collection, Hoffmann et al. developed *Application Heartbeats* [55], an application programming interface (API) that allows programs to communicate their target and actual performance characteristics to a controller. At initialization time (when an application begins), an application uses to the API to declare its desired "heartbeat" rate, expressed as a number of heartbeats over an application-chosen period of time. An application can also declare a desired latency between consecutive heartbeats. Subsequently, at run time, the application calls the API to issue heartbeats, which are signals of progress. A controller runs concurrently and invokes the API to monitor the application's "health," determined by how well the application is meeting its declared target heartbeat rate. Hoffmann and his colleagues describe both internal and external controllers. An internal controller is integrated with the application itself; it can, for example, direct its application to increase or decrease the quality of the application's output in order to meet the target heartbeat rate. An external controller is separate from the application that it controls. An example is a system resource allocator, which might grant or revoke resources from the controlled application so that it meets its target heartbeat rate. Hoffmann et al. demonstrate that the Application Heartbeats API is useful for adaptively controlling applications in isolation (to meet self-imposed rates of progress) and in combination (when resources must be brokered).

The CPU Broker and the Application Heartbeats API solve similar problems, and there are numerous parallels between the two systems. Both allow real-time applications to communicate their needs to a controller. The CPU Broker elements communicate in terms of CPU reservations; the Application Heartbeats API is also based on reservations of a sort—heartbeats per period—although the CPU time required per heartbeat is left unspecified. Both systems also attempt to be noninvasive. The CPU Broker uses delegates and advocates to decouple monitoring from the core of an application; the Application Heartbeats API is designed to require minimal instrumentation within applications.

There are two primary ways in which the CPU Broker and the Application Heartbeats system differ. First, whereas the CPU Broker is generally focused on a single, central controller, the Application Heartbeats API is essentially a blackboard architecture. Implementations of the API are simply databases of performance data; they support multiple controllers as easily as they support one.[10] The second difference lies in the actions that controllers may take. In the CPU Broker, a policy interacts with an RTOS to adjust the resources available to a controlled task. The notion of a controller in the Application Heartbeats API is somewhat more general, encompassing both internal and external adaptation as described above. In terms of the Application Heartbeats API, the CPU Broker is an external controller only. The CPU Broker does not preclude internally adaptive applications, but the architecture does not directly support them, either. (Section 5.3.3 describes research that has combined the CPU Broker with other quality-of-service mechanisms.)

Both of these differences suggest that the Application Heartbeats API can be more general-purpose than the CPU Broker. However, the CPU Broker still has certain advantages. Significantly, its architecture is more clearly centered on *prediction* rather than data collection. An advocate in the CPU Broker can be driven not only by feedback but also by application-specific knowledge of future CPU needs. Of course, when using Application Heartbeats, a custom controller can also be driven by application-specific knowledge. Unlike the CPU Broker, however, the Application Heartbeats architecture does not provide abstractions (like advocates) that clearly correspond to this concern.

Cucinotta et al. also investigated the problem of collecting scheduling-relevant data for legacy applications [17, 19]. In particular, they developed an approach for automatically inferring the values needed to make good CPU reservations for periodic multimedia applications, such as audio and video decoders and players. A CPU reservation can be defined by two time values: a period (T) and an amount of CPU time (Q) that is to be made available during each instance of the period. To infer T, Cucinotta and his colleagues developed tools that trace the system calls made by an application. (The 2010 version of the system-call tracer [19] is more efficient than the version published in 2009 [17].) Each tracing tool makes a log of system-call events and the times at which they occur. One then performs a frequency analysis of this log to estimate the period T of the traced application. To infer Q, Cucinotta and his colleagues run the application under a *feedback scheduler* that estimates the application's actual CPU demand per period.

[10]The CPU Broker can support multiple policies within a single broker, but nevertheless, the policy objects reside in a centralized controller.

Cucinotta et al. present the results of experiments to demonstrate that their technique can work and that it imposes minimal run-time overhead. They also present a theoretical analysis to show the importance of determining applications' periods correctly.

Cucinotta compare their self-tuning scheduler to the CPU Broker and point out that their approach, based on system-call monitoring, is less invasive than the CPU Broker [19]. They do not mention that the CPU Broker supports process advocates that work with unmodified legacy applications (Section 4.4.2), just as Cucinotta's scheduler does.

A more central contribution of Cucinotta's self-tuning scheduler is that it infers the periods of legacy applications. In this aspect, the work of Cucinotta and his colleagues is complementary to the work presented in Chapter 4. The CPU Broker requires that applications, through their advocates, declare their periods as part of making reservation requests. As described in Section 4.4.2, this information is obtained either through delegates or outside knowledge. The approach of Cucinotta et al., which performs frequency analysis on traces of events from unmodified periodic tasks, represents a third alternative for determining an application's period.

Historic performance information is another useful source of information for systems that, like the CPU Broker, need to accurately predict the future CPU needs of real-time tasks. To explore the use of historic performance measures for making CPU reservations, Anastasi et al. developed QoSDB [9], a repository that makes past performance data available to an online task-scheduling system. A QoSDB instance is a persistent database of performance measures. The measures for a given task are associated with an operating *mode*, which describes the context or environment for an invocation of that task. Anastasi et al. give an example: for a task that operates on image data, a useful mode value might be the size of the (uncompressed) input image in bits. For each mode, QoSDB stores a vector of performance samples and a vector of statistics, e.g., the task's average and maximum observed run times. The QoSDB interface provides functions for adding samples to the database, updating statistics, and retrieving statistics. If one requests statistics for a mode that has no historical data, QoSDB will generate and return predictions based on the data stored for other modes.

Anastasi and his colleagues performed an experiment in which they used QoSDB to create CPU reservations for tasks within a Web server. In their experimental server, the reservation's CPU time was predicted by QoSDB and the reservation's period was set to a fixed value. (The actual reservations were made by `mod_reserve`, a module that uses

AQuoSA to make CPU reservations for tasks within the Apache Web server. AQuoSA was described in Section 5.3.1.) The results showed that predictions based on historic data were accurate and therefore useful for their test application. By setting reservations according to the observed average run time of a task, Anastasi et al. could tune their system to make CPU allocations with little waste—i.e., with little over-allocation of CPU time to tasks. By setting reservations according to the maximum observed run time, the researchers could tune their system to avoid deadline misses.

Anastasi et al. referenced the CPU Broker and stated that their work with QoSDB was complementary to the kind of adaptive scheduling that the CPU Broker performs: "our proposal can be used for supporting feedback scheduling by clearly separating the feedback algorithm from the data management" [9, page 2]. This is correct, and the appropriate way to introduce QoSDB-like historic knowledge into the CPU Broker architecture is through advocates (Section 4.3.1).

The projects described above seek to improve adaptive, real-time scheduling through improved data gathering. Other work is focused on improving adaptive scheduling systems through the application of control theory. For example, Maggio and her colleagues developed an adaptive CPU scheduling system for Linux based on a mathematically modeled regulator [68, 69]. In their implementation, each real-time task (e.g., a multimedia application) provides its target and actual performance characteristics to a monitor using the Application Heartbeats API, described above. The monitor uses a *dead-beat controller*—a kind of control loop—to decide how it should adjust the CPU allocation of the task in order to drive it toward meeting its desired heartbeat rate. Maggio et al. implemented their system for multicore computer systems, and their controller decides how many cores to assign to a task during a given scheduling window. To change the heartbeat rate of a task, the controller changes the number of cores assigned to the task. The magnitude of the adjustment is based on the assumption that the performance of a task is proportional to the square root of the number of CPU cores assigned to it.

Maggio and her colleagues evaluated their scheduler for controlling isolated tasks [69] and for controlling multiple, concurrent tasks [68]. For isolated tasks, they performed experiments with tasks that fit their control model (i.e., with performance proportional to the square of the number of assigned cores), tasks that did not fit the model, and tasks with time-varying workloads. For controlling multiple, concurrent tasks, Maggio et al. discuss and evaluate five different strategies for dealing with overload—i.e., what the tasks' controllers should do when it is not possible to satisfy the CPU demands of all the tasks simultaneously. They conclude that no single strategy is ideal in all circumstances [68, page 1]:

> Our suggestions and conclusions are that a priority based mechanism should be used when real time guarantees are essential and it is possible not to accept an incoming task or application to maintain the desired performance levels (heart rates) for the other jobs present in the system. In the case of a less strict constraint on performance levels and when one wants to accept all the incoming requests, a centroid based solution may be preferable.

The main contribution of Maggio et al. is the design and evaluation of an adaptive, feedback-driven scheduler based on control theory; this is complementary to the work presented in Chapter 4, which is not based on formal control theory. The CPU Broker would obviously support policies based on control theory, however. Moreover, a central idea of the CPU Broker is that it supports user-chosen policies, including multiple policies for dealing with overload, and it allows new policies to be created through composition. The architecture of the CPU Broker thus addresses the concern expressed above, that no single policy for resolving overload is best in all circumstances.

Like Maggio and her colleagues, Abeni et al. also apply control theory in their work on feedback-driven, adaptive reservations [2]. The architectural similarities between Abeni's scheduling system and the CPU Broker were previously discussed in Section 5.3.1; the point to remember here is that Abeni's scheduling system associates each real-time task with a manager. A manager implements a control loop that monitors the performance of its task and manipulates the task's CPU reservation in order to minimize the differences between the task's recurring, periodic deadlines and the "virtual finishing times" [3] of the task's jobs. A job is one execution of the recurring unit of work within a task, and the difference between a job's deadline and its virtual finishing time is called *scheduling error*. Abeni and his colleagues present three control schemes for managers: *invariant based control* seeks to keep scheduling error within a small region of values, *stochastic dead-beat control* seeks to set the next scheduling error to zero, and *cost-optimal control* seeks to minimize a cost function of the scheduling error and CPU bandwidth. (CPU bandwidth is the amount of time available to the task within each period of a CPU reservation.) As previously noted, Abeni et al. cite the CPU Broker as an "interesting proposal" at the middleware level [2, page 133]: this is a reference to the architectures and implementations of their scheduling system and the CPU Broker. Chapter 4 does not formally analyze control schemes in the way that Abeni and his colleagues do in their publications [2, 22]. As already noted, however, the CPU Broker does provide a flexible framework for deploying policies, including policies derived from control theory.

Cucinotta and his colleagues have continued to develop the theory of feedback-driven scheduling systems based on adaptive reservations [18, 20, 21, 74], often citing the published version of Chapter 4 as related work. A primary focus of these publications is their development and analysis of control schemes that are based on formal theory. The ongoing work in this area indicates that the problems addressed by the CPU Broker—and also the software variability mechanisms that may be implemented by adaptive, feedback-driven, reservation-based scheduling—continue to be relevant to the design of computing systems.

5.3.3 Managing Multiple Kinds of Resources

The CPU Broker also influenced research toward systems that manage multiple kinds of resources simultaneously. In contrast to the CPU Broker, which manages only CPU time, these systems coordinate the allocation of multiple computing resources to applications—for example, CPU time, memory, disk access, and network access. The ability to manage multiple resources in a coordinated manner is important because most tasks ultimately depend on the availability of multiple resources in order to achieve quality of service. Multi-resource managers are also suitable for coordinating system-level adaption—i.e., the provisioning of computing resources—with application-level adaption. Application-specific qualities such as algorithm selection and output quality become resources that a system-level manager can control, and trade off against other resource types or levels of allocation, in order to maximize the QoS of a computing system as a whole. In addition, the ability to control both computing and communication resources—e.g., CPU and network access—is key for managers that seek to enable "end-to-end QoS" for distributed, real-time systems.

The FRSH/FORB framework by Sojka et al. [95] is an example of systems that seek to manage multiple kinds of resources for distributed, real-time applications. FRSH is an application programming interface for obtaining computing resources: a client makes a request for one or more resources in the form of a *contract*, and if the request is granted, the client receives a set of *virtual resources* that correspond to reservations for physical resources. FORB is a lightweight, CORBA-like middleware layer that Sojka and his colleagues use to implement a *contract broker*, which implements the FRSH API. The contract broker is a mediator between applications and resources: it receives multi-resource contract requests from applications, interacts with the resource-management services of underlying operating systems, and implements policies for optimizing the overall QoS of the set of applications that run within the distributed system. The broker is distributed, implemented by a set of agents that run on all of the nodes within the distributed system.

The general architecture of Sojka's FRSH/FORB framework is thus quite similar to that of the CPU Broker. The FRSH/FORB contract broker is implemented atop middleware, like the CPU Broker; both brokers act as mediators between applications and resources; both obtain resources for applications by interacting with the facilities of an underlying operating system; and both are based on resource reservations. Both brokers allocate CPU reservations to applications. However, as Sojka et al. note in their discussion of related work [95], the CPU Broker manages only CPU reservations. The FRSH/FORB framework, in contrast, handles requests for multiple kinds of resources. The FRSH/FORB framework is also distributed, unlike the CPU Broker, because it seeks to manage computation and communication resources for distributed applications.

Sojka and his colleagues implemented controls for three kinds of resources within their FRSH/FORB framework: CPU time, disk bandwidth, and wireless network bandwidth. For CPU reservations they used AQuoSA, described previously in Section 5.3.1. For disk reservations they used a proportional-share disk scheduler based on the *budget fair queuing* algorithm [98], and for network reservations they implemented a custom medium-access protocol called the *FRSH Wireless Protocol* (apparently also known as the *FRESCOR WLAN Protocol*) [94]). Sojka et al. evaluated their framework by demonstrating its ability to guarantee resources to a distributed video monitoring application, and thereby allow the application to maintain quality of service in the face of competing CPU, disk, and network workloads. Section 4.5.3 presents a similar experiment for the CPU Broker, for a competing CPU workload only.

Sharma et al. [90] also developed a framework for managing multiple resources—a framework that incorporated the CPU Broker directly. As described in Section 4.4.2, the CPU Broker was implemented to connect to CORBA-based applications via QuO [108], a framework for monitoring and controlling QoS in middleware-based programs. QuO was chosen because, as stated in Section 4.2, "QuO provides an excellent basis for coordinating multiple QoS management strategies in middleware-based systems, both for different levels of adaptation and for different resource dimensions." Indeed, at the time that the work in Chapter 4 was published, the authors of QuO were already applying their system in combination with the CPU Broker and a network-reservation mechanism to show that a distributed, real-time UAV simulation could be protected from competing CPU and network loads (Section 4.2). This work continued after the publication of the CPU Broker paper, yielding a new model for integrating multiple-resource QoS mechanisms into middleware-based applications: a model that Sharma et al. refer to as *qosket components*.

Qosket components are a mechanism for integrating QoS monitoring points and controls into component-based middleware applications. Both QuO and qosket components are based on interposition that is transparent to the application being controlled. However, whereas QuO delegates interpose on communication between the middleware platform (i.e., the ORB) and individual middleware-defined objects, qosket components interpose on communication between the middleware-defined components that make up an application [90]. This difference means that, in comparison to QuO delegates, qosket components are more readily deployed into component-based systems such as those built atop the industry-standard CORBA Component Model (CCM). Sharma et al. stated that their implementation of the qosket component framework includes components for network management, data shaping, and CPU management. The last of these was implemented atop the CPU Broker.

Manghwani et al. subsequently described a multi-layer architecture for "end-to-end QoS management in DRE [distributed, real-time, embedded] systems" [70, page 1] that they implemented through qosket components. *End-to-end QoS* refers to the fact that any guarantee of performance for a distributed application must depend on *coordinated* resource management across all of the devices that constitute the distributed system, from the information producers to the information consumers. Coordination means not only cooperation across individual computing devices, but also the appropriate allocation of resources of different types. As Manghwani et al. stated [70, page 2]:

> Eliminating a bottleneck by providing additional resources might simply expose a different bottleneck elsewhere that must also be managed. For example, allocating more bandwidth (e.g., using bandwidth reservation, differentiated services, alternate paths, or reducing the other load on the network) might simply expose that there isn't enough available CPU at a node along the end-to-end path to process the now plentiful data.

Manghwani et al. proposed a three-level, hierarchical software architecture for managing end-to-end QoS concerns within DRE systems. The most abstract layer, the *system layer*, is driven by two models: first, a specification of the real-time system's overall requirements, and second, a description of the system's participants and resources. From these, the system layer creates policies for system-wide resource allocation. The second architectural layer is the *local layer*. Software agents at the local layer receive the policy created by the system layer, and they translate that policy into QoS-related steps that are performed on the individual parts of the distributed system. The third layer, the *mechanism layer*, contains agents that implement QoS-relevant controls and monitoring. This includes mechanisms for allocating resources to applications (e.g., CPU time) and also mechanisms for adapting an application's behavior to better suit available resources (e.g., output rate throttling). Like all adaptive systems for resource management, Manghwani's three-level architecture is feedback-driven. Configuration data is passed down from the system layer to the local layer, and from the local layer to the mechanism layer. In the opposite direction, run-time monitoring and status data are passed upward between levels of the hierarchy.

Manghwani et al. showed how to realize their architecture via qosket components. The basic technique is as described previously: by inserting qosket components at appropriate points within the assembly of an application's ordinary components, a system designer integrates QoS controls into the application. Manghwani et al. build on this approach to deploy qosket components that correspond to the levels of their architecture: a *System Resource Manager* component, one *Local Resource Manager* component for each node of the distributed system, and multiple *QoS Mechanism* components that either (1) monitor and control node-local resources or (2) implement application-specific adaptive behaviors. The CPU-management qosket component in their implementation is an encapsulation of the CPU Broker, as previously described by Sharma et al. [90]. Manghwani's contribution is the addition of system-wide and local resource managers, which drive the CPU Broker in cooperation with controllers for network resources and application output quality.

Schantz et al. [88] subsequently provided more detail about the implementation of the QoS-management architecture described above. They evaluated the implementation by performing an experiment in which fourteen simulated UAVs transmitted image data to a set of simulated ground stations and control centers. The control centers ran image-processing software to detect targets in received images, and each time a target was recognized, the software issued an alert. (This experiment is therefore similar to, but more complex than, the experiment described in Section 4.5.3.) The total amount of image data in this experiment was sufficient to overload the network resources in the system and the CPU resources available for target recognition.

To measure the impact of end-to-end QoS management, Schantz and his colleagues configured their QoS framework to attempt to sustain the performance of a particular image pipeline—representing a case in which the data from a particular UAV is known to be more important than the data from the other UAVs. Their QoS framework ensured that the designated stream's network traffic was prioritized over competing traffic; in addition, the framework used the CPU Broker to reserve CPU time for processing the designated stream's images. As a result, the designated stream was able to perform well in their experiment, as measured by the number of frames successfully received, the latency of received images, and the number of targets detected. Schantz et al. concluded that their "multi-layer middleware-mediated QoS framework that integrates resource management mechanisms" [88, page 1204] allowed the critical portion of their multi-UAV distributed system to achieve both higher performance and better predictability than it would otherwise have achieved [88].

The capstone flight demonstration of the DARPA Program Composition for Embedded Systems (PCES) program, which supported the development of the CPU Broker, was also based on a multi-UAV surveillance and target-tracking scenario. Unlike the software simulations described above and in Section 4.5.3, the capstone flight demonstration involved actual UAVs and related military equipment. As described by Loyall et al. [66], the PCES capstone demonstration was designed to exemplify the issues and challenges of developing DRE applications that maintain high end-to-end QoS characteristics. It was also designed to showcase PCES-supported technologies that address those challenges. Within the overall demonstration, the CPU Broker was expected to provide CPU guarantees at the ground station for the UAVs [66, page 91]:

> The ground station will be extended with an extra processor that will be running PCES developed software (C++) to do image processing and QoS management. This ground station processor will also be running the TimeSys Linux real-time OS, with a PCES-developed CPU Broker for CPU management and Differentiated Services for network management. PCES developed QoS management software will be retrieving the analog imagery from the ScanEagle ground station receiver, digitizing it, and shaping it (e.g., changing its rate, size, and resolution) to fit the need of the C2 node and the capacity of the network and CPU resources available.

The CPU Broker's role in the PCES demonstration was therefore similar to its role in the experiment presented in Section 4.5.3.

Ultimately, the CPU Broker was utilized in dry runs of the demonstration but not in the final event. Loyall and Schantz explained that TimeSys Linux, the operating system that underlies the implementation of the CPU Broker, was not quite up to the task [67, page 36-27]: “. . . stability problems in the underlying Timesys Linux RT operating system caused us to omit the Timesys Linux RT CPU control capability for the final capstone demonstration (although we used it for shorter dry runs) and revert to using the more limited Linux OS priorities instead.” This was unfortunate, but apparently not due to a problem with the CPU Broker. The CPU Broker’s role is to provide mediated access to the CPU-reservation facilities of an underlying operating system—not to implement reservation-based scheduling itself.

In addition to presenting the outcome of the PCES capstone demonstration, Loyall and Schantz [67] summarize the development and application of end-to-end QoS-management systems based on QuO and qosket components. Interested readers may wish to use this summary as a guide to the other publications described previously in this section.

As described in Section 1.4.3, the CPU Broker is a software variability mechanism that allows the tasks of a real-time system to be composed late in the software life cycle, after the individual tasks have been implemented and delivered to a system designer. It supports late composition through its ability to connect to managed tasks in a noninvasive fashion. Schantz and Loyall state that these qualities are desirable for QoS-management systems in general: they write that a well-designed, reusable, and maintainable QoS management “dovetails with and reinforces extended software engineering practices” [87, page 151] including separation of concerns, component-based designs, and service-oriented designs. The CPU Broker and the qosket component architecture are both based on these principles.

Looking forward, Schantz and Loyall identify several issues that should be addressed in future research and development efforts toward platforms that provide multiple-resource and end-to-end QoS management for DRE systems [87]. These issues demand advances throughout the software development life cycle: the continued evolution of methods that combine and analyze multiple QoS aspects, i.e., multiple resources and adaption strategies; libraries of reusable QoS mechanisms, management components, and policies; tools to automate the design process and assist with the evaluation of controlled systems; and conflict identification and resolution across QoS dimensions. The CPU Broker-related work presented in this chapter has addressed some, but not all of these concerns. Resource brokers like the CPU Broker are building blocks—and only building blocks—for the design of self-adaptive systems. Schantz and Loyall are correct that additional research will be necessary if the vision of autonomic computing [59] and self-adaptive systems [86], assembled from “off the shelf” software parts, is to be more fully realized.

5.4 References

[1] Abeni, L., and Buttazzo, G. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA '99)* (Hong Kong, China, Dec. 1999), pp. 70–77.

[2] Abeni, L., Cucinotta, T., Lipari, G., Marzario, L., and Palopoli, L. QoS management through adaptive reservations. *Real-time Systems 29*, 2–3 (Mar. 2005), 131–155.

[3] Abeni, L., Palopoli, L., Lipari, G., and Walpole, J. Analysis of a reservation-based feedback scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS '02)* (Austin, TX, Dec. 2002), pp. 71–80.

[4] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer 1986 USENIX Conf.* (June 1986), pp. 93–112.

[5] Aigner, R. Development of an IDL compiler. Grosser Beleg, Dresden University of Technology, Jan. 2001.

[6] Aigner, R. Development of an IDL compiler for micro-kernel based components. Master's thesis, Dresden University of Technology, Sept. 2001.

[7] Aigner, R. *Communication in Microkernel-Based Operating Systems*. PhD thesis, Technische Universität Dresden, July 2010.

[8] Aigner, R. DICE project homepage. `http://www.inf.tu-dresden.de/index.php?node_id=1432`, Feb. 2010.

[9] Anastasi, G. F., Cucinotta, T., Lipari, G., and García-Valls, M. A QoS registry for adaptive real-time service-oriented applications. In *Proc. of the 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)* (Irvine, CA, Dec. 2011).

[10] Apel, S., Leich, T., Rosenmüller, M., and Saake, G. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Generative Programming and Component Engineering*, R. Glück and M. Lowry, Eds., vol. 3676 of *Lecture Notes in Computer Science*. Springer, 2005, pp. 125–140.

[11] Auerbach, J., Barton, C., Chu-Carroll, M., and Raghavachari, M. Mockingbird: Flexible stub compilation from pairs of declarations. In *Proc. of the 19th International Conference on Distributed Computing Systems (ICDCS)* (Austin, TX, May–June 1999), pp. 393–402.

[12] Baumann, A. Inter-dispatcher communication in Barrelfish. Barrelfish Technical Note 011, ETH Zurich, Dec. 2011.

[13] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. Lightweight remote procedure call. *ACM Transactions on Computer Systems 8*, 1 (Feb. 1990), 37–55.

[14] Chakravarthy, V., Regehr, J., and Eide, E. Edicts: Implementing features with flexible binding times. In *Proc. of the 7th International Conference on Aspect-Oriented Software Development (AOSD)* (Brussels, Belgium, Mar.–Apr. 2008), pp. 108–119.

[15] Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., Volanschi, E.-N., Lawall, J., and Noyé, J. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys 30*, 3es (Sept. 1998).

[16] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. A generic component model for building systems software. *ACM Transactions on Computer Systems 26*, 1 (Feb. 2008), 1–42.

[17] Cucinotta, T., Abeni, L., Palopoli, L., and Checconi, F. The wizard of OS: A heartbeat for legacy multimedia applications. In *Proc. of the IEEE/ACM/IFIP 7th Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)* (Grenoble, France, Oct. 2009), pp. 70–79.

[18] Cucinotta, T., Abeni, L., Palopoli, L., and Lipari, G. A robust mechanism for adaptive scheduling of multimedia applications. *ACM Transactions on Embedded Computing Systems 10*, 4 (Nov. 2011).

[19] Cucinotta, T., Checconi, F., Abeni, L., and Palopoli, L. Self-tuning schedulers for legacy real-time applications. In *Proc. of the 5th European Conference on Computer Systems (EuroSys)* (Paris, France, Apr. 2010), pp. 55–68.

[20] Cucinotta, T., and Palopoli, L. Feedback scheduling for pipelines of tasks. In *Hybrid Systems: Computation and Control*, A. Bemporad, A. Bicchi, and G. Buttazzo, Eds., vol. 4416 of *Lecture Notes in Computer Science*. Springer, 2007, pp. 131–144.

[21] Cucinotta, T., and Palopoli, L. QoS control for pipelines of tasks using multiple resources. *IEEE Transactions on Computers 59*, 3 (Mar. 2010), 416–430.

[22] Cucinotta, T., Palopoli, L., Marzario, L., Lipari, G., and Abeni, L. Adaptive reservations in a Linux environment. In *Proc. of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (Toronto, ON, May 2004), pp. 238–245.

[23] Douglass, B. P. *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*. Newnes/Elsevier, 2011.

[24] Eide, E., Lepreau, J., and Simister, J. L. Flexible and optimized IDL compilation for distributed applications. In *Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR '98)*, D. O'Hallaron, Ed., vol. 1511 of *Lecture Notes in Computer Science*. Springer, May 1998, pp. 288–302.

[25] Eide, E., Simister, J. L., Stack, T., and Lepreau, J. Flexible IDL compilation for complex communication patterns. *Scientific Programming 7*, 3, 4 (1999), 275–287.

[26] Fassino, J.-P., Stefani, J.-B., Lawall, J., and Muller, G. Think: A software framework for component-based operating system kernels. In *Proc. of the USENIX Annual Technical Conference* (Monterey, CA, June 2002), pp. 73–86.

[27] Feske, N. A case study on the cost and benefit of dynamic RPC marshalling for low-level system components. *Operating Systems Review 41*, 4 (July 2007), 40–48.

[28] Filman, R. E., Elrad, T., Clarke, S., and Akşit, M. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[29] FitzRoy-Dale, N. Magpie software. `http://ertos.nicta.com.au/software/kenge/magpie/latest/`, Jan. 2006.

[30] FitzRoy-Dale, N. A declarative approach to extensible interface compilation. In *Proc. of the 1st International Workshop on Microkernels for Embedded Systems (MIKES)* (Sydney, Australia, Jan. 2007), pp. 43–49.

[31] FitzRoy-Dale, N. *Architecture Optimisation*. PhD thesis, University of New South Wales, Mar. 2010.

[32] FitzRoy-Dale, N., Kuz, I., and Heiser, G. Architecture optimisation with Currawong. In *Proc. of the 1st ACM Asia-Pacific Workshop on Systems (APSys)* (New Delhi, India, Aug. 2010), pp. 7–12.

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

[34] Flatt, M., and Felleisen, M. Units: Cool units for HOT languages. In *Proc. of the ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation (PLDI)* (Montreal, Canada, June 1998), pp. 236–248.

[35] Flux Research Group. *Flick: The Flexible IDL Compiler Kit Version 2.1: Programmer's Manual*. University of Utah Department of Computer Science, Nov. 1999. `http://www.cs.utah.edu/flux/flick/current/doc/guts/guts.ps.gz`.

[36] Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., and Shivers, O. The Flux OSKit: A substrate for OS and language research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles* (St. Malo, France, Oct. 1997), pp. 38–51.

[37] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. Microkernels meet recursive virtual machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation* (Seattle, WA, Oct. 1996), USENIX Assoc., pp. 137–151.

[38] Gabber, E., Small, C., Bruno, J., Brustoloni, J., and Silberschatz, A. The Pebble component-based operating system. In *Proc. of the USENIX Annual Technical Conference* (Monterey, CA, June 1999), pp. 267–282.

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

[40] Garlan, D., and Shaw, M. An introduction to software architecture. Tech. Rep. CMU–CS–94–166, Carnegie Mellon University School of Computer Science, Jan. 1994.

[41] Gay, D., Levis, P., and Culler, D. Software design patterns for TinyOS. *ACM Transactions on Embedded Computing Systems 6*, 4 (Sept. 2007).

[42] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)* (San Diego, CA, June 2003), pp. 1–11.

[43] Gefflaut, A., Jaeger, T., Park, Y., Liedtke, J., Elphinstone, K. J., Uhlig, V., Tidswell, J. E., Deller, L., and Reuther, L. The SawMill multiserver approach. In *Proc. of the 9th SIGOPS European Workshop* (Kolding, Denmark, Sept. 2000), pp. 104–109.

[44] Genssler, T., Nierstrasz, O., Schönhage, B., Christoph, A., Winter, M., Ducasse, S., Wuyts, R., Arévalo, G., Müller, P., and Stich, C. Components for embedded software: The PECOS approach. In *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (Grenoble, France, Oct. 2002), pp. 19–26.

[45] Gokhale, A., and Schmidt, D. C. Evaluating the performance of demultiplexing strategies for real-time CORBA. In *Proc. of the IEEE Global Telecommunications Conference (GLOBECOM)* (Phoenix, AZ, Nov. 1997), pp. 1729–1734.

[46] Gokhale, A., and Schmidt, D. C. Techniques for optimizing CORBA middleware for distributed embedded systems. In *Proc. of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)* (New York, NY, Mar. 1999), pp. 513–521.

[47] Gokhale, A. S., and Schmidt, D. C. Optimizing a CORBA Internet Inter-ORB Protocol (IIOP) engine for minimal footprint embedded multimedia systems. *IEEE Journal on Selected Areas in Communications 17*, 9 (Sept. 1999), 1673–1706.

[48] Greenstein, B., Kohler, E., and Estrin, D. A sensor network application construction kit (SNACK). In *Proc. of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)* (Baltimore, MD, Nov. 2004), pp. 69–80.

[49] Grisby, D. *omniidl — The omniORB IDL Compiler*, June 2000. `http://omniorb.sourceforge.net/omni41/omniidl.pdf`.

[50] Grisby, D., Lo, S.-L., and Riddoch, D. *The omniORB version 4.1 User's Guide*, July 2009. `http://omniorb.sourceforge.net/omni41/omniORB.pdf`.

[51] Haeberlen, A. Using platform-specific optimizations in stub-code generation. Study thesis, System Architecture Group, University of Karlsruhe, Germany, July 2002.

[52] Haeberlen, A., Liedtke, J., Park, Y., Reuther, L., and Uhlig, V. Stub-code performance is becoming important. In *Proc. of the 1st Workshop on Industrial Experiences with Systems Software (WIESS)* (San Diego, CA, Oct. 2000), pp. 31–38.

[53] Hauck, F. J., Becker, U., Geier, M., Meier, E., Rastofer, U., and Steckermeier, M. AspectIX: A quality-aware, object-based middleware architecture. In *New Developments in Distributed Applications and Interoperable Systems*, K. Zielinski, K. Geihs, and A. Laurentowski, Eds., vol. 70 of *IFIP*. Springer, 2002, pp. 115–120.

[54] Helander, J., and Forin, A. MMLite: A highly componentized system architecture. In *Proc. of the 8th SIGOPS European Workshop* (Sintra, Portugal, Sept. 1998), pp. 96–103.

[55] Hoffmann, H., Eastep, J., Santambrogio, M. D., Miller, J. E., and Agarwal, A. Application Heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Proc. of the 7th International Conference on Autonomic Computing (ICAC)* (Washington, DC, June 2010), pp. 79–88.

[56] Kalogeraki, V., Melliar-Smith, P. M., Moser, L. E., and Drougas, Y. Resource management using multiple feedback loops in soft real-time distributed object systems. *The Journal of Systems and Software 81*, 7 (July 2008), 1144–1162.

[57] Kell, S. Rethinking software connectors. In *Proc. of the International Workshop on Synthesis and Analysis of Component Connectors (SYANCO)* (Dubrovnik, Croatia, Sept. 2007), pp. 1–12.

[58] Kell, S. Component adaptation and assembly using interface relations. In *Proc. of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)* (Reno, NV, Oct. 2010), pp. 322–340.

[59] Kephart, J. O., and Chess, D. M. The vision of autonomic computing. *IEEE Computer 36*, 1 (Jan. 2003), 41–50.

[60] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming*, J. L. Knudsen, Ed., vol. 2072 of *Lecture Notes in Computer Science*. Springer, 2001, pp. 327–353.

[61] Kono, K., and Masuda, T. Efficient RMI: Dynamic specialization of object serialization. In *Proc. of the 20th International Conference on Distributed Computing Systems (ICDCS)* (Taipei, Taiwan, Apr. 2000), pp. 308–315.

[62] Kuz, I., Liu, Y., Gorton, I., and Heiser, G. CAmkES: A component model for secure microkernel-based embedded systems. *The Journal of Systems and Software 80*, 5 (May 2007), 687–699.

[63] L4Ka Project. IDL4 compiler. `http://www.l4ka.org/projects/idl4/`, Nov. 2003.

[64] Lafortune, E. ProGuard. `http://proguard.sourceforge.net/`.

[65] Levis, P., Gay, D., Handziski, V., Hauer, J.-H., Greenstein, B., Turon, M., Hui, J., Klues, K., Sharp, C., Szewczyk, R., Polastre, J., Buonadonna, P., Nachman, L., Tolle, G., Culler, D., and Wolisz, A. T2: A second generation OS for embedded sensor networks. Tech. Rep. TKN–05–007, Telecommunication Networks Group, Technische Universität Berlin, Nov. 2005.

[66] Loyall, J., Schantz, R., Corman, D., Paunicka, J., and Fernandez, S. A distributed real-time embedded application for surveillance, detection, and tracking of time critical targets. In *Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (San Francisco, CA, Mar. 2005), pp. 88–97.

[67] Loyall, J. P., and Schantz, R. E. Dynamic QoS management in distributed real-time embedded systems. In *Handbook of Real-Time and Embedded Systems*, I. Lee, J. Y.-T. Leung, and S. H. Son, Eds., Chapman & Hall/CRC Computer and Information Science Series. Chapman & Hall/CRC, 2008.

[68] Maggio, M., Hoffmann, H., Agarwal, A., and Leva, A. Control-theoretical CPU allocation: Design and implementation with feedback control. In *6th International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID)* (Karlsruhe, Germany, June 2011).

[69] Maggio, M., Hoffmann, H., Santambrogio, M. D., Agarwal, A., and Leva, A. Controlling software applications via resource allocation within the Heartbeats framework. In *Proc. of the 49th IEEE Conference on Decision and Control (CDC)* (Atlanta, GA, Dec. 2010), pp. 3736–3741.

[70] Manghwani, P., Loyall, J., Sharma, P., Gillen, M., and Ye, J. End-to-end quality of service management for distributed real-time embedded applications. In *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Denver, CO, Apr. 2005).

[71] Muller, G., Marlet, R., Volanschi, E.-N., Consel, C., Pu, C., and Goel, A. Fast, optimized Sun RPC using automatic program specialization. In *Proc. of the 18th International Conference on Distributed Computing Systems (ICDCS)* (Amsterdam, The Netherlands, May 1998), pp. 240–249.

[72] Muller, G., Volanschi, E.-N., and Marlet, R. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *Proc. of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)* (Amsterdam, The Netherlands, June 1997), pp. 116–126.

[73] Open Software Foundation and Carnegie Mellon University. *Mach 3 Server Writer's Guide*. Cambridge, MA, Jan. 1992.

[74] Palopoli, L., Cucinotta, T., Marzario, L., and Lipari, G. AQuoSA—adaptive quality of service architecture. *Software—Practice and Experience 39*, 1 (Jan. 2009), 1–31.

[75] Pandey, R., and Wu, J. BOTS: A constraint-based component system for synthesizing scalable software systems. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems (LCTES)* (Ottawa, Canada, June 2006), pp. 189–198.

[76] Petersen, A. Patterns in C — part 1. *C Vu 17*, 1 (Feb. 2005).

[77] Petersen, A. Patterns in C — part 2: STATE. *C Vu 17*, 2 (Apr. 2005).

[78] Petersen, A. Patterns in C — part 3: STRATEGY. *C Vu 17*, 3 (June 2005).

[79] Petersen, A. Patterns in C — part 4: OBSERVER. *C Vu 17*, 4 (Aug. 2005).

[80] Petersen, A. Patterns in C — part 5: REACTOR. *C Vu 17*, 5 (Oct. 2005).

[81] Prehofer, C. Feature-oriented programming: A fresh look at objects. In *ECOOP '97 — Object-Oriented Programming*, M. Aksit and S. Matsuoka, Eds., vol. 1241 of *Lecture Notes in Computer Science*. Springer, 1997, pp. 419–443.

[82] Reid, A., Flatt, M., Stoller, L., Lepreau, J., and Eide, E. Knit: Component composition for systems software. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, Oct. 2000), pp. 347–360.

[83] Reiser, H., Steckermeier, M., and Hauck, F. J. IDLflex: A flexible and generic compiler for CORBA IDL. Technical Report TR–I4–01–08, Friedrich-Alexander-University Informatik 4, Erlangen-Nürnberg, Germany, Sept. 2001.

[84] Rosenmüller, M., Siegmund, N., Apel, S., and Saake, G. Flexible feature binding in software product lines. *Automated Software Engineering 18*, 2 (June 2011), 163–197.

[85] Rosenmüller, M., Siegmund, N., Saake, G., and Apel, S. Code generation to support static and dynamic composition of software product lines. In *Proc. of the 7th International Conference on Generative Programming and Component Engineering (GPCE)* (Nashville, TN, Oct. 2008), pp. 3–12.

[86] Salehie, M., and Tahvildari, L. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems 4*, 2 (May 2009).

[87] Schantz, R., and Loyall, J. Composing and decomposing QoS attributes for distributed real-time systems: Experience to date and hard problems going forward. In *Composition of Embedded Systems: Scientific and Industrial Issues*, F. Kordon and O. Sokolsky, Eds., vol. 4888 of *Lecture Notes in Computer Science*. Springer, 2007, pp. 150–167.

[88] Schantz, R. E., Loyall, J. P., Rodrigues, C., and Schmidt, D. C. Controlling quality-of-service in distributed real-time and embedded systems via adaptive middleware. *Software—Practice and Experience 36*, 11–12 (Sept.–Oct. 2006), 1189–1208.

[89] Shapiro, J., and Miller, M. CapIDL language specification: Version 0.1. `http://www.coyotos.org/docs/build/capidl.pdf`, Feb. 2006.

[90] Sharma, P. K., Loyall, J. P., Heineman, G. T., Schantz, R. E., Shapiro, R., and Duzan, G. Component-based dynamic QoS adaptations in distributed real-time and embedded systems. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, vol. 3291 of *Lecture Notes in Computer Science*. Springer, 2004, pp. 1208–1224.

[91] Shenoy, P., Hasan, S., Kulkarni, P., and Ramamritham, K. Middleware versus native OS support: Architectural considerations for supporting multimedia applications. In *Proc. of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (San Jose, CA, Sept. 2002), pp. 23–32.

[92] Smaragdakis, Y., and Batory, D. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology 11*, 2 (Apr. 2002), 215–255.

[93] Smith, J. M. *Elemental Design Patterns*. Addison-Wesley, 2012.

[94] Sojka, M., Molnár, M., Trdlička, J., Jurčík, P., Smolík, P., and Hanzálek, Z. Wireless networks — documented protocols, demonstration. Technical Report, Deliverable D–ND3v2, version 1.0/Final, Czech Technical University, Dec. 2008. Available at `http://www.frescor.org/index.php?page=publications`.

[95] Sojka, M., Píša, P., Faggioli, D., Cucinotta, T., Checconi, F., Hanzálek, Z., and Lipari, G. Modular software architecture for flexible reservation mechanisms on heterogeneous resources. *Journal of Systems Architecture 57*, 4 (Apr. 2011), 366–382.

[96] Swint, G. S., Pu, C., Jung, G., Yan, W., Koh, Y., Wu, Q., Consel, C., Sahai, A., and Koichi, M. Clearwater: Extensible, flexible, modular code generation. In *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Long Beach, CA, Nov. 2005), pp. 144–153.

[97] Uhlig, V. A micro-kernel-based multiserver file system and development environment. Tech. Rep. RC21582, IBM T. J. Watson Research Center, Oct. 1999.

[98] Valente, P., and Checconi, F. High throughput disk scheduling with fair bandwidth distribution. *IEEE Transactions on Computers 59*, 9 (Sept. 2010), 1172–1186.

[99] van Ommering, R. Building product populations with software components. In *Proc. of the 24th International Conference on Software Engineering (ICSE)* (Orlando, FL, May 2002), pp. 255–265.

[100] van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. The Koala component model for consumer electronics software. *IEEE Computer 33*, 3 (Mar. 2000), 78–85.

[101] Vergnaud, T., Hugues, J., Pautet, L., and Kordon, F. PolyORB: A schizophrenic middleware to build versatile reliable distributed applications. In *Reliable Software Technologies — Ada-Europe 2004*, A. Llamosí and A. Strohmeier, Eds., vol. 3063 of *Lecture Notes in Computer Science*. Springer, 2004, pp. 106–119.

[102] Welling, G., and Ott, M. Customizing IDL mappings and ORB protocols. In *Middleware 2000: IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, J. Sventek and G. Coulson, Eds., vol. 1795 of *Lecture Notes in Computer Science*. Springer, Apr. 2000, pp. 396–414.

[103] Wils, A. *An Architecture-Centric Approach for Developing Timing-Driven Self-Adaptive Software Systems*. PhD dissertation, Katholieke Universiteit Leuven, Mar. 2007.

[104] Zalila, B., Hugues, J., and Pautet, L. An improved IDL compiler for optimizing CORBA applications. In *Proc. of the 2006 Annual ACM SIGAda International Conference on Ada* (Albuquerque, New Mexico, Nov. 2006), pp. 21–28.

[105] Zhang, C., Gao, D., and Jacobsen, H.-A. Towards just-in-time middleware architectures. In *Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD)* (Chicago, IL, Mar. 2005), pp. 63–74.

[106] Zhang, C., and Jacobsen, H.-A. Re-factoring middleware systems: A case study. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, vol. 2888 of *Lecture Notes in Computer Science*. Springer, 2003, pp. 1243–1262.

[107] Zhang, C., and Jacobsen, H.-A. Resolving feature convolution in middleware systems. In *Proc. of the 19th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Vancouver, BC, Oct. 2004), pp. 188–205.

[108] Zinky, J. A., Bakken, D. E., and Schantz, R. D. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems 3*, 1 (1997), 55–73.

CHAPTER 6

CONCLUSION

Through the analysis of three examples, this dissertation provides strong support for the claim that *it is feasible and useful to realize variability mechanisms in novel ways to improve the run-time performance of software systems.* Each example is based on a well-known style of modular software composition. Each implements a variability mechanism in a nonstandard fashion, while maintaining implementation modularity, to address the performance issues that are associated with stereotypical implementations of the mechanism.

The first example, presented in Chapter 2, demonstrates that remote procedure call (RPC) and remote method invocation (RMI) mechanisms can be improved through a novel approaches to interface definition language (IDL) compilation. The Flick IDL compiler kit allows a software developer to *compose the compiler* to produce the RPC or RMI stubs that are suited to a particular application. The use of appropriate intermediate representations within Flick enables both flexibility and domain-specific performance optimizations during code generation.

The second example, detailed in Chapter 3, is a novel approach to realizing design patterns in software that is made from statically instantiated and connected components. The approach separates the static parts of a software design from the dynamic parts of the system's behavior, allowing a designer to *compose pattern participants* at the level of component assemblies. The separation makes a pattern-based software implementation more amenable to analysis. This in turn can enable more effective and domain-specific detection of system design errors, better prediction of run-time behavior, and more effective optimization.

The third example details the run-time performance improvements obtained by a novel implementation of task scheduling in multi-agent real-time systems. Chapter 4 presents the design and evaluation of a new CPU Broker, which mediates between the CPU demands of a set of real-time processes and the facilities of a real-time operating system. The broker modularizes CPU-management and allows a system designer or configurer to *compose the* *policy* for a multi-agent system as a whole. As demonstrated in Chapter 4, the CPU Broker can help a multi-agent system maintain quality of service in the face of events such as changes in CPU availability or changes in the importances of the managed tasks.

Software developers should be encouraged by these examples. Their message is that the general understanding of variability mechanisms is incomplete, and that the study of variability mechanisms can yield significant benefits. Many important variability mechanisms, such as code generation and design patterns, are underutilized in practice because people confuse the *concepts* of the mechanisms with their stereotypical *implementations*. The applicability and consequences of the implementations become confused with those of the concepts in general. This dissertation shows that a nontraditional implementation of a well-known variability mechanism can allow that mechanism to be applied in novel ways to solve problems in the design and implementation of configurable software products.

A significant area of future research is the further development and evaluation of new realizations for well-known variability mechanisms. The examples in this dissertation focus on realizations that enhance the run-time performance of software. Alternative realizations might also focus on performance, or they might focus on other different concerns such as static analyzability (for the absence of configuration errors) or dynamic fault tolerance (for the containment of configuration errors).

A second area of future research is the development of new variability mechanisms. Such mechanisms may involve any of the artifacts of a software system's implementation—e.g., build scripts, compilers and similar tools, version-control repositories, operating systems, and so on—not just a system's basic source code. Because of this, the space for new variability mechanisms is extremely broad. Within the wide space of potential approaches, a particularly interesting and fruitful area for future research is based on new programming-language constructs that can be used for variation management in software product lines. Aspect-oriented programming is one example of recent language-based research that has important application to software product lines. Feature-oriented programming is another. Language-based or otherwise, the invention of new variability mechanisms may help future software designers and implementers manage variation and configuration is ways that are not possible or commonly practiced today.

Software systems today are increasingly made from configurable collections of software parts; the implementations of the configuration choices are instances of variability mechanisms. This dissertation has explained the benefits gained through novel realizations of three well-known mechanisms. As future software systems are made from ever larger and more complex assemblies of parts and configuration choices, the need for effective variability mechanisms will grow. The work presented in this dissertation can be seen as initial research toward a deeper understanding of variability mechanisms, an understanding that will be increasingly necessary for the design and construction of software systems in the future.