Specialization of CML message-passing primitives

John Reppy
University of Chicago
jhr@cs.uchicago.edu

Yingqi Xiao
University of Chicago
xiaoyq@cs.uchicago.edu

Abstract

Concurrent ML (CML) is a statically-typed higher-order concurrent language that is embedded in Standard ML. Its most notable feature is its support for *first-class synchronous operations*. This mechanism allows programmers to encapsulate complicated communication and synchronization protocols as first-class abstractions, which encourages a modular style of programming where the underlying channels used to communicate with a given thread are hidden behind data and type abstraction.

While CML has been in active use for well over a decade, little attention has been paid to optimizing CML programs. In this paper, we present a new program analysis for statically-typed higher-order concurrent languages that enables the compile-time specialization of communication operations. This specialization is particularly important in a multiprocessor or multicore setting, where the synchronization overhead for general-purpose operations are high. Preliminary results from a prototype that we have built demonstrate that specialized channel operations are much faster than the general-purpose operations.

Our analysis technique is modular (*i.e.*, it analyzes and optimizes a single unit of abstraction at a time), which plays to the modular style of many CML programs. The analysis consists of three steps: the first is a type-sensitive control-flow analysis that uses the program's type-abstractions to compute more precise results. The second is the construction of an *extended control-flow graph* using the results of the CFA. The last step is an iterative analysis over the graph that approximates the usage patterns of known channels. Our analysis is designed to detect special patterns of use, such as one-shot channels, fan-in channels, and fan-out channels. We have proven the safety of our analysis and state those results.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages — Program analysis; D.3.2 [*Programming Languages*]: Language classifications — Concurrent, distributed, and parallel languages; D.3.3 [*Programming Languages*]: Language Constructs and Features — Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors — Optimization

General Terms Languages, Performance

Keywords ML, concurrent languages, message passing, static analysis

© ACM (2007). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *The Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (January 2007).

Copyright © 2007 ACM 1-59593-575-4/07/0001…$5.00.

1. Introduction

Concurrent ML (CML) [Rep91, Rep99] is a statically-typed higher-order concurrent language that is embedded in Standard ML [MTHM97]. CML extends SML with synchronous message passing over typed channels and a novel abstraction mechanism, called *first-class synchronous operations*, for building synchronization and communication abstractions. This mechanism allows programmers to encapsulate complicated communication and synchronization protocols as first-class abstractions, which encourages a modular style of programming where the actual underlying channels used to communicate with a given thread are hidden behind data and type abstraction. CML has been used successfully in a number of systems, including a multi-threaded GUI toolkit [GR93], a distributed tuple-space implementation [Rep99], and a system for implementing partitioned applications in a distributed setting [YYS+01]. The design of CML has inspired many implementations of CML-style concurrency primitives in other languages. These include other implementations of SML [MLt], other dialects of ML [Ler00], other functional languages, such as HASKELL [Rus01], SCHEME [FF04], and our own MOBY language [FR99], and other high-level languages, such as JAVA [Dem97].

While CML has been in active use for well over a decade, little attention has been paid to optimizing CML programs. In this paper, we present a new program analysis for statically-typed higher-order concurrent languages that is a significant step toward optimization of CML. Our technique is modular (*i.e.*, it analyzes and optimizes a single unit of abstraction at a time), which plays to the modular style of many CML programs. The analysis consists of three steps. The first is a new twist on traditional control-flow analysis (CFA) that we call *type-sensitive* CFA [Rep06]. This analysis is a modular 0-CFA that tracks values of abstract type (*i.e.*, types defined in the module that are abstract outside the module) that escape "into the wild." Because of type abstraction, we known that any value of an abstract type that comes in from the wild must have previously escaped from the module. The second step is construction of an *extended control-flow graph* (CFG) from the result of the CFA. This extended CFG has extra edges to represent process creation, values communicated by message-passing, and values communicated via the outside world (a.k.a. the wild). The last step is an iterative analysis of the CFG, which computes an approximation of the number of processes that send or receive messages on the channel, as well as an approximation of the number of messages sent on the channel. This information allows us to detect special patterns of use (or topologies), such as one-shot channels, fan-in channels, and fan-out channels. These special patterns can then be exploited by using more efficient implementations of channel primitives.

The paper has the following organization. In the next section, we describe specific patterns of communication that are common in message-passing programs. We also describe channel operations that are specialized to these patterns and which have measurably

better performance than the general-purpose ones. We also present an example of a prototypical server as is found in many CML applications and use it to illustrate the opportunities for specialized communication. In Section 3, we define the small concurrent language that we use to present our analysis and we give a dynamic semantics for it. This semantics has the property that it explicitly tracks the execution history of individual processes; we use these execution histories to characterize the dynamic properties of channels that must be guaranteed to safely use the specialized forms. The main technical content of the paper is the presentation of our analysis, which we break up into four sections. In Section 4, we present the type-sensitive CFA for our language. This analysis is defined for a single unit of abstraction and its result allows us to characterize a subset of the defined channels as *known channels*; *i.e.*, channels whose send and receive sites are all statically known. We then present the construction of the extended CFG in Section 5. The edges in this graph are labeled with the set of known channels that are live across the edge. In Section 6, we describe the analysis of the CFG that results in an approximation of the module's communication topology and the static properties that allow safe specialization of communication primitives. The final technical section outlines the proofs of correctness for our analysis (full proofs can be found in the second author's Master's paper [Xia05]). We discuss related work in Section 8 and the implementation status and future work in Section 9. Finally we conclude in Section 10.

2. Specialization of communication primitives

Synchronous channels are the main communication and synchronization mechanism of CML. The underlying protocols used to implement these channels are necessarily general, since they must function correctly and fairly in arbitrary contexts. Despite this generality, the existing implementation of CML is very efficient with minimal overhead on concurrent operations [Rep99]. We believe that this efficiency is important because it fosters a programming style that uses thread abstraction freely without incurring a debilitating performance cost. One of the main reasons for this efficiency is that there is only one underlying thread executing at any time, which allows the implementation to easily single-thread critical regions.

With the advent of inexpensive desktop multiprocessors and with multicore processors appearing even in laptops, there is a real need to port CML to a multiprocessor environment. Unfortunately, implementing the CML primitives in a multiprocessor environment incurs significant additional runtime overhead, since more complicated protocols are required. The main motivation of the research described in this paper is to develop compiler optimizations that can reduce this overhead.

As might be expected, most uses of CML primitives fall into one of a number of common patterns, which are amenable to more efficient implementation. As is often the case, the hard part of this optimization technique is developing an effective, but efficient, analysis that identifies when it is safe to specialize. Furthermore, we want this analysis to be modular so that it will easily scale to larger systems. Fortunately, CML's design emphasizes a modular programming style based on user-defined concurrency abstractions. While the motivation for this programming style is to promote more robust software, it also provides an opportunity for optimization, since the abstraction provided by first-class synchronous operations allows modular analysis to determine the communication topology.

2.1 Common patterns of communication

Assuming that the basic communication primitive is a synchronous channel, we consider the following possible communication patterns:

	number of		
senders	**receivers**	**messages**	**topology**
≤ 1	≤ 1	≤ 1	one-shot
≤ 1	≤ 1	>1	one-to-one
≤ 1	>1	>1	one-to-many (fan-out)
>1	≤ 1	>1	many-to-one (fan-in)
>1	>1	>1	many-to-many

In this table, the notation >1 denotes the possibility that more than one thread or message may be involved and the notation ≤ 1 denotes that at most one thread or message is involved. For example, the one-to-one pattern involves arbitrary numbers of messages, but at most one sender and receiver. The many-to-many pattern is the general case. An analysis of message-passing patterns is *safe* if whenever it approximates the number of messages of threads as ≤ 1, then that property holds for all possible executions. It is always safe to return an approximation of >1.

Another dimension of interest is whether a channel is used in choice contexts, since there is additional overhead in the implementation of channels to support fairness and negative acknowledgments in choice contexts. A channel that is not used in choice contexts can have a simpler, and more efficient, implementation. For the bulk of this paper, we restrict ourselves to a language without choice, but we describe how our analysis can be extended to handle choice in Section 9.

While programmers could specialize implementations by hand, doing so would complicate the programming model and could lead to less reliable software. Furthermore, correctness of a given protocol often depends on the properties of the primitives used to implement it. Changes to the protocol may require changes in the choice of primitives, which makes the protocol harder to maintain. For these reasons, we believe that an automatic optimization technique based on program analysis and compiler transformations is necessary.

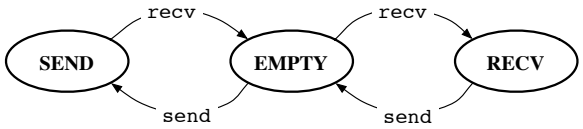
2.2 Specialized channel operations

In general, a CML channel must support communication involving multiple sending and receiving processes transmitting multiple messages in arbitrary contexts. This generality requires a complicated protocol to implement with commiserate overhead.[1] In a multiprocessor setting, the protocol used to implement channel communication involves multiple locks and other overhead. On the other hand, if we know that a channel has a restricted pattern of use, then we can design a more efficient implementation.

For example, consider a one-to-one channel; *i.e.*, a channel that has at most one sender and one receiver thread at any time and is not used in a choice context. Such a channel can only be in one of three distinct states:

EMPTY — neither thread is waiting for communication,
RECV — the receiver thread is waiting for the sender, or
SEND — the sender thread is waiting for the receiver.

Furthermore, the channel's usage pattern means that its state transitions are restricted to the following state diagram:



Thus, the `send` operation can be implemented under the assumption that the channel's state must be either **EMPTY** or **RECV**,

[1] Chapter 10 of *Concurrent Programming in ML* describes CML's implementation, while Knabe has described a similar protocol in a distributed setting [Kna92].

```
structure SimpleServ : SIMPLE_SERV =
  struct
    datatype serv = S of (int * int chan) chan

    fun new () = let
          val ch = channel()
          fun server v = let
                val (req, replCh) = recv ch
                in
                  send(replCh, v);
                  server req
                end
          in
            spawn (server 0);
            S ch
          end

    fun call (S ch, v) = let
          val replCh = channel()
          in
            send (ch, (v, replCh));
            recv replCh
          end
  end
```

Figure 1. A simple service with an abstract client-server protocol

which means that a single atomic compare-and-swap instruction can be used to test for the **EMPTY** state and, if **EMPTY**, set the state to **SEND**. If the state was not **EMPTY**, then it must be **RECV** and the `send` operation can be completed without further synchronization.[2]

To understand the benefits of specialized channel operations in the multiprocessor setting, we have developed a prototype implementation of CML channel operations that includes specialized implementations for the patterns described in the previous section. This prototype is written in C and assembly language and we have tested it on both a dual single-core system and a quad dual-core system. While a detailed description of this prototype is beyond the scope of this paper, our preliminary results show that the specialized channels are significantly faster than the general-purpose channels. For example, the one-to-one channel described above is 3-4 times faster than the general-purpose channel in our multiprocessor implementation and is as fast as the single-threaded CML channel on the same hardware.

2.3 An example

To illustrate how the analysis and optimization might proceed, consider the simple service implemented in Figure 1. This service has the following abstract interface:[3]

```
signature SIMPLE_SERV =
  sig
    type serv
    val new : unit -> serv
    val call : (serv * int) -> int
  end
```

The `new` function creates a new instance of the service by allocating a new channel and spawning a new server thread to handle requests on the channel. The representation of the service is the request channel, but it is presented as an abstract type. The

[2] Of course, there is also the need to schedule the receiver thread for execution, but that cost would be required no matter how the channel protocol is implemented.

[3] To keep the example concise, we use direct operations on channels instead of CML's event operations, but event values can be handled without difficulty (see Section 9).

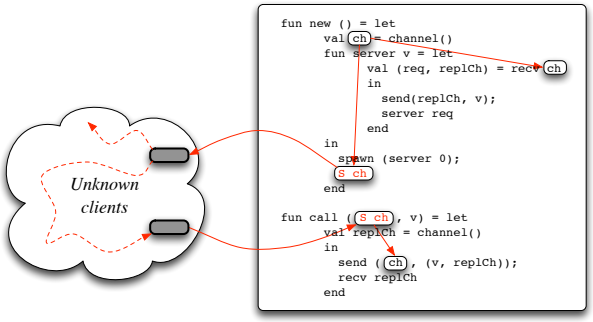


Figure 2. Data-flow of the server's request channel

```
structure SimpleServ : SIMPLE_SERV =
  struct
    datatype serv
      = S of (int * int OneShot.chan) FanIn.chan

    fun new () = let
          val ch = FanIn.channel()
          fun server v = let
                val (req, replCh) =
                        FanIn.recv ch
                in
                  OneShot.send(replCh, v);
                  server req
                end
          in
            spawn (server 0);
            S ch
          end

    fun call (S ch, v) = let
          val replCh = OneShot.channel()
          in
            FanIn.send (ch, (v, replCh));
            OneShot.recv replCh
          end
  end
```

Figure 3. A version of Figure 1 with specialized communication operations

`call` function sends a request to a given instance of the service. The request message consists of the request and a fresh channel for the reply. Because the connection to the service is represented as an abstract type, we know that even though it escapes out of the `SimpleServ` module, it cannot be directly accessed by unknown code. Figure 2 illustrates the data-flow of the service's request channel. Specifically, we observe the following facts:

- For a given instance of the service, the request channel has a many-to-one communication pattern.
- For a given client request, the reply channel has a one-to-one communication pattern and is used at most once (*i.e.*, it is a one-shot channel).

We can exploit these facts to specialize the communication operations resulting in the optimized version of the service shown in Figure 3. We have highlighted the specialized code and have assumed the existence of a module `FanIn` that implements channels specialized for the many-to-one pattern and a module `OneShot` that

is specialized for one-shot channels. In our prototype implementation, the specialized version of this service has 60% higher throughput than the version implemented using general-purpose channels. While the relative benefit for a more computationally intensive service would be less, minimizing communication overhead encourages the use of thread abstraction for encapsulating light-weight state.

Because of the signature ascription, we know all of the send and receive sites for the `ch` and `replCh` channels, but if we added the function

```
fun reveal (S ch) = ch
```

to the service's interface, then the above transformation would no longer be safe, since clients could use the `reveal` function to gain access to the server's request channel and use it to send and receive messages in ways not supported by the specialized channels. The technical challenge is to develop program analyses that can detect the patterns described in Section 2.1 automatically when they are present, but also recognize the situation where access to the channel is not limited (as with the `reveal` function). Another issue that the analysis must address is distinguishing between multiple threads that are created at the same spawn point. For example, say we have

```
fun twice f = (f(); f())
```

and we create two servers sharing a common request channel using the code

```
twice (fn () => spawn(server 0));
```

Then our analysis should detect that the request channel `ch` is not a fan-in channel. Note, however, that `replCh` is still a one-shot channel.

3. A concurrent language

We present our algorithm in the context of a small statically-typed concurrent language. This language is a monomorphic subset of Core SML [MTHM97] with explicit types and concurrency primitives. Standard ML and other ML-like languages use modules to organize code and signature ascription to define abstraction, but use the **abstype** declaration to define abstractions in lieu of modules. We further simplify this declaration form to only have a single data constructor. Figure 4 gives the abstract syntax for this simple language. A program p is a sequence of zero or more **abstype** declarations followed by an expression. The analysis that we present below is modular and can be applied to each **abstype** declaration (d) independently. Each **abstype** definition defines a new abstract type (T) and corresponding data constructor (D) and a collection of functions (fb_i). Outside the **abstype** declaration, the type T is abstract (*i.e.*, the data constructor D is not in scope). The sequential expression forms include let-bindings, nested function bindings, function application, data-constructor application and deconstruction,[4] and pair construction and projection. In addition, there are four concurrent expression forms: channel definition, process spawning, message sending, and message receiving. Types include abstract types (T), function types, pair types, and channel types. Abstract types are either predefined types (*e.g.*, `unit`, `int`, `bool`, *etc.*) or are defined by an **abstype** declaration.

This language does not include CML's event types or the corresponding event combinators, but based on experience with our prototype implementation, we believe that it is straightforward to add these to the analysis framework, so we omit them to keep the presentation more compact.

[4] In a language with sum types, deconstruction would be replaced by a case expression.

$$
\begin{array}{rcl}
p & ::= & e \\
 & | & d\,p \\
d & ::= & \textbf{abstype}\ T\ =\ D\ \textbf{of}\ \tau\ \textbf{with}\ fb_1\ \cdots\ fb_n\ \textbf{end} \\
fb & ::= & \textbf{fun}\ f\,(x)\ =\ e \\
e & ::= & x \\
 & | & \bullet \\
 & | & \textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2 \\
 & | & \textbf{fun}\ f\,(x)\ =\ e_1\ \textbf{in}\ e_2 \\
 & | & e_1\ e_2 \\
 & | & D\ e \\
 & | & \textbf{let}\ D\,x\ =\ e_1\ \textbf{in}\ e_2 \\
 & | & \langle e_1,\ e_2\rangle \\
 & | & \#i\,e \qquad \text{where } i \in \{1,2\} \\
 & | & \textbf{chan}\ c\ \textbf{in}\ e \\
 & | & \textbf{spawn}\ e \\
 & | & \textbf{send}(e_1,\ e_2) \\
 & | & \textbf{recv}\ e \\
\tau & ::= & T \\
 & | & \tau_1 \rightarrow \tau_2 \\
 & | & \tau_1 \times \tau_2 \\
 & | & \textbf{chan}\,\tau
\end{array}
$$

Figure 4. A simple concurrent language

We assume that variables, data-constructor names, and abstract-type names are globally unique. We also assume that variables and constructors are annotated with their type. We omit this type information most of the time for the sake of brevity, but, when necessary, we write it as a superscript (*e.g.*, x^τ). One should think of this language as a compiler's intermediate representation following typechecking.

We use LVAR to denote the set of variables defined in the current **abstype** declaration, GVAR to denote variables defined elsewhere, and VAR = LVAR ∪ GVAR for all variables defined or mentioned in the program. We denote the known function identifiers by FUNID ⊂ LVAR (*i.e.*, those variables that are defined by function bindings) and the known channel identifiers by CHANID ⊂ LVAR (*i.e.*, those variables that are defined by channel bindings). The set ABSTY is the set of abstract type names and DATACON is the set of data constructors.

3.1 Dynamic semantics

Following Colby [Col95], the semantics for our language tracks execution history on a per-process basis. This information is necessary to characterize the dynamic usage of channels. Since **abstype** declarations do not play a rôle in the dynamic semantics of the language, we think of a program as a sequence of nested function bindings. For example,

$$
\begin{array}{l}
\textbf{abstype}\ T\ =\ D\ \textbf{of}\ \tau\ \textbf{with} \\
\quad \textbf{fun}\ f\,(x)\ =\ e_1 \\
\quad \textbf{fun}\ g\,(y)\ =\ e_2 \\
\textbf{end} \\
e_3
\end{array}
$$

is treated as

$$\textbf{fun}\ f\,(x)\ =\ e_1\ \textbf{in}\ \textbf{fun}\ g\,(y)\ =\ e_2\ \textbf{in}\ e_3$$

For a given program p, we assume that each expression in p is labeled with a unique program point $a \in$ PROGPT. We write $a : e$ to denote that e is the expression at program point a. Furthermore,

we assume that for each $a \in \text{PROGPT}$, there is a $\bar{a} \in \text{PROGPT}$. The $\bar{a}$ labels are not used to label expressions, but serve to distinguish between parent and child threads in control paths. A *control path* is a finite sequence of program points: $\text{CTLPATH} = \text{PROGPT}^*$. We use π to denote an arbitrary control path and juxtaposition to denote concatenation. We say that $\pi \preceq \pi'$ if π is a prefix of π'. Control paths are used to uniquely label dynamic instances of channels, which we write $c@\pi$, where $c \in \text{CHANID}$. We also use k to denote dynamic channel values, and K to denote the set of dynamic channel values.

Evaluation of the sequential features of the language follows a standard small-step presentation based on evaluation contexts [FF86]. We modify the syntax of expression terms to distinguish *values* as follows:

$$\begin{array}{rcl} v & ::= & \bullet \\ & | & (\mathbf{fun}\ f\ (x)\ =\ e) \\ & | & k \\ & | & \langle v_1,\, v_2 \rangle \\ \\ e & ::= & v \\ & | & \cdots \end{array}$$

The unit value ($\bullet$) was already part of the syntax, but we add function values, dynamic channel values, and pairs of values. With these definitions, we can define the sequential evaluation relation $e \rightsquigarrow e'$ by the rules in Figure 5. Evaluation contexts are defined in the standard call-by-value way and are used in the definition of concurrent evaluation.

$$\begin{array}{rcl} E & ::= & [\,] \\ & | & \mathbf{let}\ x\ =\ E\ \mathbf{in}\ e \mid \mathbf{let}\ D\,x\ =\ E\ \mathbf{in}\ e \\ & | & E\ e \mid v\ E \mid D\ E \\ & | & \mathbf{send}(E,\, e) \mid \mathbf{send}(v,\, E) \mid \mathbf{recv}\ E \\ & | & \langle E,\, e \rangle \mid \langle v,\, E \rangle \mid \#i\,E \end{array}$$

For the semantics of concurrent evaluation, we represent the state of a computation as a tree, where the nodes of the tree are labeled with expressions representing process states and edges are labeled with the program point corresponding to the evaluation step taken from the parent to the child. The leaves of the tree represent the current states of the processes in the computation. Because a tree captures the history of the computation as well as its current state, we call it a *trace*. Nodes in a trace are uniquely named by control paths that describe the path from the root to the node. In defining traces, it is useful to view them as prefix-closed finite functions from control paths to expressions. If t is a trace, then we write $t.\pi$ to denote the node one reaches by following π from the root, and if $t.\pi$ is a leaf of t, a is a program point, and e an expression, then $t \cup \{\pi a \mapsto e\}$ is the trace with a child e added to $t.\pi$ with the new edge labeled by a. For a program p, the initial trace will be the map $\{\epsilon \mapsto p\}$, where ϵ is the empty control path.

We define concurrent evaluation as the smallest relation ($\Rightarrow$) between traces satisfying the following four rules. The first rule lifts sequential evaluation to traces.

$$\frac{t.\pi = E[a : e] \text{ is a leaf} \quad e \rightsquigarrow e'}{t \Rightarrow t \cup \{\pi a \mapsto E[e']\}}$$

The second rule deals with channel creation.

$$\frac{t.\pi = E[a : \mathbf{chan}\ c\ \mathbf{in}\ e] \text{ is a leaf}}{t \Rightarrow t \cup \{\pi a \mapsto E[e[c \mapsto c@\pi a]]\}}$$

The third rule deals with process creation.

$$\frac{t.\pi = E[a : \mathbf{spawn}\ e] \text{ is a leaf}}{t \Rightarrow t \cup \{\pi a \mapsto E[\bullet], \pi \bar{a} \mapsto e\}}$$

The last rule deals with communication.

$$\frac{\begin{array}{c} t.\pi_1 = E_1[a_1 : \mathbf{send}(k,\, v)] \text{ is a leaf} \\ t.\pi_2 = E_2[a_2 : \mathbf{recv}\ k] \text{ is a leaf} \end{array}}{t \Rightarrow t \cup \{\pi_1 a_1 \mapsto E_1[\bullet], \pi_2 a_2 \mapsto E_2[v]\}}$$

The set of traces of a program represents all possible executions of the program. It is defined as

$$\text{Trace}(p) = \{t \mid \{\epsilon \mapsto p\} \Rightarrow^* t\}$$

3.2 Properties of traces

Let p be a program and let c be a channel identifier in p. For any trace $t \in \text{Trace}(p)$ and $k = c@\pi$ occurring in t, we define the dynamic send and receive sites of k as follows:

$$\begin{array}{rcl} \text{Sends}_t(k) & = & \{\pi \mid t.\pi = E[\mathbf{send}(k,\, v)]\} \\ \text{Recvs}_t(k) & = & \{\pi \mid t.\pi = E[\mathbf{recv}\ k]\} \end{array}$$

We say that c has the *single-sender* property if for any $t \in \text{Trace}(p)$, $k = c@\pi$ occurring in t, and $\pi_1, \pi_2 \in \text{Sends}_t(k)$, either $\pi_1 \preceq \pi_2$ or $\pi_2 \preceq \pi_1$. The intuition here is that if $\pi_1 \preceq \pi_2$ then the sends can not be concurrent. On the other hand, if π_1 and π_2 are not related by $\preceq$, then they may be concurrent.[5] Note that the single-sender property allows multiple processes to send messages on a given channel, they are just not allowed to do it concurrently. Likewise, we say that c has the *single-receiver* property if for any $t \in \text{Trace}(p)$, $k = c@\pi$ occurring in t, and $\pi_1, \pi_2 \in \text{Recvs}_t(k)$, either $\pi_1 \preceq \pi_2$ or $\pi_2 \preceq \pi_1$.

We can now state the special channel topologies from Section 2.1 as properties of the set of traces of a program. For a channel identifier c in a program p, we can classify its topology as follows:

- The channel c is a *one-shot* channel if for any $t \in \text{Trace}(p)$ and $k = c@\pi$ occurring in t, $|\text{Sends}_t(k)| \leq 1$.
- The channel c is *point-to-point* if it has both the single-sender and single-receiver properties.
- The channel c is a *fan-out* channel if it has the single-sender property, but not the single-receiver.
- The channel c is a *fan-in* channel if it has the single-receiver property, but not the single-sender.

Our analysis computes safe approximations of these properties, which are described in Section 6.1.

4. Type-sensitive control-flow analysis for CML

The first step of our analysis is a standard abstract-interpretation-style control-flow analysis of the program [Shi91, Ser95]. The goal of this analysis is two-fold: first we need to determine the control-flow and data-flow of the program, but we also want to identify *known channels* (*i.e.*, channels for which we know the creation site and all use sites). One might use a whole-program analysis for this purpose, but we have developed a modular CFA instead. This CFA is based on Serrano's version of 0-CFA [Ser95], but with a couple of important differences. First, our source language is statically typed and has concurrency operations. Second, our analysis exploits the type abstraction in the program, such as provided by ML signature ascription or **abstype** definitions, to improve the quality of the results. We call our analysis "*Type-sensitive CFA*." A detailed description of this algorithm can be found in a recent paper [Rep06], but we sketch the technique here.

The basic intuition behind our approach is that if a value's type is abstract outside the scope of a module, then any value of that type can only be allocated by code that is inside the module and

[5] There may be other causal dependencies, such as synchronizations, that would order π_1 and π_2, but our model does not take these into account.

$$
\begin{array}{rcl}
\textbf{let}\ x\ =\ v\ \textbf{in}\ e & \rightsquigarrow & e[x \mapsto v] \\
\textbf{let}\ D\,x\ =\ D\,v\ \textbf{in}\ e & \rightsquigarrow & e[x \mapsto v] \\
\textbf{fun}\ f\,(x)\ =\ e_1\ \textbf{in}\ e_2 & \rightsquigarrow & e_2[f \mapsto (\textbf{fun}\ f\,(x)\ =\ e_1)] \\
(\textbf{fun}\ f\,(x)\ =\ e)\ v & \rightsquigarrow & e[f \mapsto (\textbf{fun}\ f\,(x)\ =\ e), x \mapsto v] \\
\#i\,\langle v_1,\, v_2\rangle & \rightsquigarrow & v_i
\end{array}
$$

Figure 5. Sequential evaluation

any operation on the value's representation must also be inside the module. We reflect this intuition by computing a mapping from the abstract types of a module to an approximation of the values of that type that have escaped into the wild. The analysis then uses this mapping to approximate any unknown values of abstract type that might flow into the module. Note that our approach should apply to any language that has data abstraction mechanisms. In the remainder of this section, we discuss those aspects of our CFA that are important to the analysis of CML code.

4.1 Approximate values

The analysis computes a mapping from variables to approximate values (ABSVAL), which are given by the following grammar:

$$
\begin{array}{rcl}
\widehat{v} & ::= & \bot \\
 & | & D\,\widehat{v} \mid \bullet \mid \langle \widehat{v}_1,\, \widehat{v}_2\rangle \\
 & | & F \mid C \\
 & | & \widehat{T} \mid \widehat{\textbf{chan}\,\tau} \mid \widehat{\tau_1 \rightarrow \tau_2} \\
 & | & \top
\end{array}
$$

where $D \in$ DATACON, $F \in 2^{\text{FUNID}}$, $C \in 2^{\text{CHANID}}$, and $T \in$ ABSTY. We use $\bot$ to denote undefined or not yet computed values, $D\,\widehat{v}$ for an approximate value constructed by applying D to $\widehat{v}$, $\langle \widehat{v}_1,\, \widehat{v}_2\rangle$ for an approximate pair, F for a set of known functions, and C for a set of known channels. Our analysis will only compute sets of functions and sets of channels where all the members have the same type (see [Rep06] for a proof of this property) and so we extend our type annotation syntax to include such sets. In addition to the single *top* value found in most presentations of CFA, we have a family of top values ($\widehat{\tau}$) indexed by type. The value $\widehat{\tau}$ represents an unknown value of type τ (where τ is either a function or abstract type). We define the $\vee$ operation in the standard way to combine two approximate values.

4.2 Type-sensitive CFA

Our analysis algorithm iteratively computes a 4-tuple of approximations: $\mathcal{A} = (\mathcal{V}, \mathcal{C}, \mathcal{R}, \mathcal{T})$, where

$\mathcal{V}$	$\in$	VAR $\rightarrow$ ABSVAL	variable approximation
$\mathcal{C}$	$\in$	CHANID $\rightarrow$ ABSVAL	channel-message approximation
$\mathcal{R}$	$\in$	FUNID $\rightarrow$ ABSVAL	function-result approximation
$\mathcal{T}$	$\in$	ABSTY $\rightarrow$ ABSVAL	escaping abstract-value approximation

Our $\mathcal{V}$ approximation corresponds to Serrano's $\mathcal{A}$; $\mathcal{C}$ is an approximation of the messages sent on a given known channel; $\mathcal{R}$ records an approximation of function results for each known function, which is used in lieu of analyzing a function's body when the function is already being analyzed, and $\mathcal{T}$ records escaping values and is used to interpret abstract values of the form $\widehat{T}$.

The main workhorse of our CFA is the function `cfa` that takes an expression e and an approximation $\mathcal{A}$ and computes the approximate value of e and a, possibly different, approximation.

4.3 Analysis of message-passing operations

When analyzing message send and receive operations, we use the $\mathcal{C}$ approximation to connect the send and receive sites for a given channel. For example, when computing$\texttt{cfa}([\![\textbf{recv}\ e]\!],\, \mathcal{A})$ we first compute

$$(\widehat{v}, \mathcal{A}') = \texttt{cfa}([\![e]\!],\, \mathcal{A})$$

If $\widehat{v}$ is of the form C, then the approximate result of the receive operation is the join of the approximate values carried by all of the channels in C:

$$\bigvee_{c \in C} \mathcal{C}(c)$$

Otherwise, $\widehat{v}$ is a top value, but if its type is $\textbf{chan}\,T$, we can use $\mathcal{T}(T)$ for the approximation of the receive operation.

For send operations, if C is the approximation of the first argument and $\widehat{v}_2$ is the approximation of the second, then we replace the $\mathcal{C}$ component of $\mathcal{A}$ with $\mathcal{C}'$, which is defined to be

$$\mathcal{C}'(c) = \begin{cases} \mathcal{C}(c) \vee \widehat{v}_2 & \text{if } c \in C \\ \mathcal{C}(c) & \text{otherwise} \end{cases}$$

In this way, the CFA is able to propagate approximations from send sites to receive sites.

4.4 Escaping abstract values

In Serrano's analysis (and any other modular CFA that we are aware of), escaping values are treated conservatively. For example, the analysis assumes that any escaping function can be called on any value, so the functions parameters are approximated as $\top$. For escaping channels, this would mean assuming arbitrary senders and receivers and arbitrary messages, which would make modular analysis of typical CML modules, such as our example, useless. To avoid this problem, our analysis tracks escaping values of abstract type by recording them in the $\mathcal{T}$ approximation. In turn, $\mathcal{T}$ is used to approximate values of abstract type that come in from the wild.

4.5 Known channels

Our CFA allows one to compute certain static approximations of the dynamic properties described in Section 3.2. Figure 6 gives the approximation of the send and receive sites for a given channel. If the channel escapes (denoted $\text{Esc}(c)$), then we use $\top$ to denote the set. A channel for which we know all of the send and receive sites is called a *known channel*.

4.6 An example

To illustrate our type-sensitive CFA, we revisit the example of Figure 1, but recast in the notation of our simple language (with a few syntactic liberties). Figure 7 shows the example with program-point labels included. Our CFA produces the following information for this example:

$$
\begin{array}{rcl}
\widehat{\text{SendSites}}(\texttt{ch}) & = & \{a_{13}\} \\
\widehat{\text{RecvSites}}(\texttt{ch}) & = & \{a_4\} \\
\widehat{\text{SendSites}}(\texttt{replCh}) & = & \{a_5\} \\
\widehat{\text{RecvSites}}(\texttt{replCh}) & = & \{a_{14}\}
\end{array}
$$

Thus, both `ch` and `replCh` are known channels.

$$\widehat{\text{SendSites}}(c) = \begin{cases} \{a \mid a : \mathbf{send}(e_1,\, e_2) \in p \wedge c \in \mathcal{A}(e_1)\} & \text{if } \neg\text{Esc}(c) \\ \top & \text{if Esc}(c) \end{cases}$$

$$\widehat{\text{RecvSites}}(c) = \begin{cases} \{a \mid a : \mathbf{recv}\ e \in p \wedge c \in \mathcal{A}(e)\} & \text{if } \neg\text{Esc}(c) \\ \top & \text{if Esc}(c) \end{cases}$$

Figure 6. Approximation of channel send and receive sites

```
a1 :  fun new () = (
a2 :    chan ch in
a3 :    fun server v = (
a4 :      let (w', replCh') = recv ch in
a5 :        send (replCh', v);
a6 :        server w')
      in
a7 :      spawn (a8 : server 0);
a9 :      S ch)

a10 :  fun call (s, w) = (
a11 :    let S ch' = s in
a12 :    chan replCh in
a13 :      send (ch', (w, replCh));
a14 :      recv replCh)
```

Figure 7. The simple service in our simple language

5. The extended CFG

With the information from the CFA in hand, the next step of our analysis is to construct an extended control-flow graph (CFG) for the module that we are analyzing. We then use this extended CFG to compute approximate trace fragments that can be used to analyze the topology of the program.

There is a node in the graph for each program point; in addition, there is an entry and exit node for each function definition. A node with a label a corresponds to the point in the program's execution where the next redux is labeled with a. The graph has four kinds of edges. The first two of these represent control flow, while the other two are used to track the flow of values outside the module.

1. *Control edges* represent normal sequential control-flow.
2. *Spawn edges* represent process creation. If there is an expression a_1 : **spawn** e and a_2 is the label of the first redux in e, then there will be a spawn edge from a_1 to a_2.
3. *Message edges* are added from send sites to receiver sites for known channels.
4. *Wild edges* are added to represent the potential flow of abstract values from one function in the module to another.

The graph is constructed such that a control edge from a_1 to a_2 corresponds to a trace edge labeled with a_1 that leads to a trace node labeled by a_2. Similarly, a spawn edge from a_1 to a_2 in the CFG corresponds to $\bar{a_1}$ in a trace. More formally, the sets of nodes and edges are defined to be

$$\begin{aligned} n \in \textsc{Node} &= \textsc{ProgPt} \cup (\textsc{FunId} \times \{\mathbf{entry}, \mathbf{exit}\}) \\ \textsc{EgLabel} &= \{\mathbf{ctl}, \mathbf{spawn}, \mathbf{msg}, \mathbf{wild}\} \\ \textsc{Edge} &= \textsc{Node} \times \textsc{EgLabel} \times \textsc{Node} \\ G \in \textsc{Graph} &= 2^{\textsc{Node}} \times 2^{\textsc{Edge}} \end{aligned}$$

The successors of a node n in a graph G are defined to be $\text{Succ}_G(n) = \{n' \mid (n, l, n') \text{ is an edge in } G\}$.

Constructing the CFG is done in three steps. First we create the basic graph with control and spawn edges in the obvious way. One important point is that we use the results of the CFA to determine the edges from call sites to known functions. Note that because we are only interested in tracking known channels, which by definition cannot have escaped the module, we can ignore calls to unknown functions when constructing the graph. Message edges are added in much the same way as control edges for known function calls. Let a : $\mathbf{send}(e_1,\, e_2)$ be a send in the program and assume that the CFA computed C as the approximation of e_1. Then for each channel $c \in C$ and $a' \in \widehat{\text{RecvSites}}(c)$, we add a send edge from a to a' to the graph. We add wild edges from any site where an abstract value escapes the module to any site where such a value can return from the wild. Once we have constructed the graph, we use a liveness analysis to label the edges with the set of known channels that are live across the edge. As described in the next section, we use these edge labels to limit the scope of the analysis on a per-channel basis.

5.1 An example

Figure 8 give the extended control-flow graph for our running example. We have labeled each edge with the set of known channels that are live across the edge. This graph illustrates the three ways that a channel can be shared among multiple threads (and thus have multiple senders/receivers):

1. A process is spawned that has the channel in its closure. This is represented by the channel being in the label of the spawn edge (e.g., `ch` on the edge from a_7 to a_8).
2. The channel is sent in a message from one process to another. This is represented by the channel being in the label of the message edge (e.g., `replCh` on the edge from a_{13} to a_4).
3. The channel escapes into the wild and then returns as the argument to an exported function. This is represented by the channel being in the label of a wild edge from the exit of one function to the entry of another (e.g., `ch` on the edge from the exit of `new` to the entry of `call`).

6. Analyzing the CFG

The final stage of our analysis involves using the CFG to determine the communication topology. We do this step independently for each known channel in the module. Because the analysis is concerned with only a single channel c at a time, we can ignore those parts of the graph where c is not live (essentially remove any edge that does not contain c in its label set). The analysis computes a finite map $\widehat{P}$ that maps program points to an approximation of the possible control paths that execution follows to get to the program point.

$$\widehat{P} \in \widehat{\textsc{PathTo}} = \textsc{ProgPt} \xrightarrow{\text{fin}} 2^{\widehat{\textsc{CtlPath}}}$$

where the set of abstract control paths is defined by the syntax

$$\begin{aligned} \widehat{\pi} \quad ::= \quad & *{:}\pi \\ \mid \quad & \pi_1{:}\pi_2 \end{aligned}$$

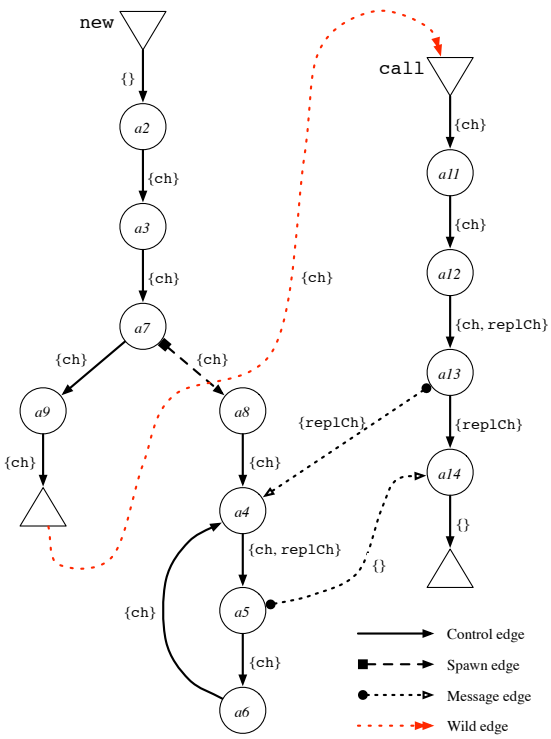


Figure 8. The CFG for the example

For an approximate control paths $\widehat{\pi}$, we split the path into a process ID part (the part before the ':') and a path. The process ID can either be '$*$', which is used to represent an unknown set of processes, or a path that uniquely identifies the process. We define an ordering $\sqsubseteq$ on abstract control paths as follows: $\pi_1{:}\pi_1' \sqsubseteq \pi_2{:}\pi_2'$ if $\pi_1 = \pi_2$ and $\pi_1' \preceq \pi_2'$. In other words, $\widehat{\pi}_1 \sqsubseteq \widehat{\pi}_2$ if they are paths in the same process and if $\widehat{\pi}_1$ is a prefix of $\widehat{\pi}_2$. The following notation is used to project the process ID part from an approximate control path:

$$\begin{aligned} \widehat{\mathrm{Proc}}(*{:}\pi_2) &= * \\ \widehat{\mathrm{Proc}}(\pi_1{:}\pi_2) &= \pi_1 \end{aligned}$$

We lift $\widehat{\mathrm{Proc}}$ to sets of control paths in the standard way. If A is a set of approximate control paths, then we define the number of distinct processes in A as follows:

$$\begin{aligned} \widehat{\mathrm{NumProcs}}(A) &= \infty \quad \text{if } * \in \widehat{\mathrm{Proc}}(A) \\ \widehat{\mathrm{NumProcs}}(A) &= |\widehat{\mathrm{Proc}}(A)| \quad \text{otherwise} \end{aligned}$$

The analysis of a CFG G is defined by a pair of mutually recursive functions:

$$\begin{aligned} \mathcal{N}_G^c &: \textsc{Node} \to \widehat{\textsc{CtlPath}} \to \widehat{\textsc{PathTo}} \to \widehat{\textsc{PathTo}} \\ \mathcal{E}_G^c &: \textsc{Edge} \to \widehat{\textsc{CtlPath}} \to \widehat{\textsc{PathTo}} \to \widehat{\textsc{PathTo}} \end{aligned}$$

The definition of these functions can be found in Figure 9, where $\widehat{P}\emptyset = \{a \mapsto \emptyset \mid a \in \textsc{ProgPt}\}$ is the finite map that assigns the empty path set to every program point.

The $\mathcal{N}_G^c$ function is defined by case analysis. For a function f's entry it follows the unique control edge to the first program point of f. For the exit of f, it computes the union of the analysis for all outgoing edges. These edges will either be control edges to f's call sites, when f is a known function, or wild edges, when f is an escaping function. For program-point nodes, we have three subcases. If the approximation $\widehat{P}$ already contains a path $pid{:}\pi_1 a \pi_2$ that precedes $\widehat{\pi}$ and $pid{:}\pi_1 \in \widehat{P}(a)$, then we have looped (the loop is $a \to \pi_2 \to a$) and can stop. If the number of processes that can reach the program point a is greater than one, then we stop.[6] Otherwise, we record the visit to a in $\widehat{P}'$ and compute the union over the outgoing edges.

The $\mathcal{E}_G^c$ function is also defined by cases. When the edge is a control edge labeled by a, we analyze the destination node by passing in the extended path $\widehat{\pi}a$. When the edge is a spawn edge, we analyze the destination node by passing in a new process ID paired with the empty path. For message edges, we analyze the receive site using the send-site program point as a new process ID. This choice of process ID distinguishes the send from other sends that target the same receive sites, but it conflates multiple receive sites that are targets of the same send, which is safe since only one receive site can actually receive the message. For wild edges, we analyze the destination node using '$*$' as the process ID, since any number of threads might call the target of the wild edge with the same dynamic instance of the channel c.

6.1 Static classification of channels

For a known channel c that is defined at a : **chan** c **in** e, we can statically classify c by examining $\widehat{P_c} = \mathcal{N}_G^c[\![a]\!]\epsilon{:}\epsilon\,\widehat{P}\emptyset$. First we define the approximate send and receive contexts for c as follows:

$$\begin{aligned} \widehat{S_c} &= \bigcup_{a \in \widehat{\mathrm{SendSites}}(c)} \widehat{P_c}(a) \\ \widehat{R_c} &= \bigcup_{a \in \widehat{\mathrm{RecvSites}}(c)} \widehat{P_c}(a) \end{aligned}$$

These are the static approximations of the Sends and Recvs sets from Section 3.2. We say that a known channel c has the *static single sender* (resp. *static single receiver*) property if it is the case that $\widehat{\mathrm{NumProcs}}(\widehat{S_c}) \leq 1$ (resp. $\widehat{\mathrm{NumProcs}}(\widehat{R_c}) \leq 1$). The static classification of channels then follows the dynamic classification from Section 3.1.

- If $\widehat{\mathrm{NumProcs}}(\widehat{S_c}) \leq 1$ and $\nexists \widehat{\pi}_1, \widehat{\pi}_2 \in \widehat{S_c}$ with $\widehat{\pi}_1 \neq \widehat{\pi}_2$ and $\widehat{\pi}_1 \sqsubseteq \widehat{\pi}_2$, then c is a one-shot channel.
- If c has both the static single-sender and static single-receiver properties, then it is a point-to-point channel.
- If c has the static single-sender property, but not the static single-receiver, then it is a *fan-out* channel.
- If c has the static single-receiver property, but not the static single-sender, then it is a *fan-in* channel.

6.2 An example

Once again, we turn to our running example to illustrate the intuition behind our analysis. Recall that we have two known channels: `ch`, which is created at a_2, and `replCh`, which is created at a_{12}.

We first consider `replCh`. Figure 10 give the restriction of the CFG from Figure 8 to the subgraph in which `replCh` is live. Notice that although `replCh` is received by the server in its loop, the fact that `replCh` is not live after node a_5 means that we do not analyze the loop and thus avoid confusing different instances of `replCh` with each other. Computing

$$\widehat{P_{\texttt{replCh}}} = \mathcal{N}_G^{\texttt{replCh}}[\![a_{11}]\!]\epsilon{:}\epsilon\,\widehat{P}\emptyset$$

[6] Recall that we are interested in channels that have *single* senders or receivers.

$$
\begin{aligned}
\mathcal{N}^c_G[\![(f,\mathbf{entry})]\!]\widehat{\pi}\,\widehat{P} &= \mathcal{N}^c_G[\![a']\!]\widehat{\pi}\,\widehat{P} \quad \text{where } \mathrm{Succ}_G(f,\mathbf{entry}) = \{a'\} \\
\mathcal{N}^c_G[\![(f,\mathbf{exit})]\!]\widehat{\pi}\,\widehat{P} &= \widehat{P} \cup \left(\bigcup_{e \in \mathrm{Edge}_G(a)} \mathcal{E}^c_G[\![e]\!]\widehat{\pi}\,\widehat{P} \right) \\
\mathcal{N}^c_G[\![a]\!]\widehat{\pi}\,\widehat{P} &= \widehat{P} \quad \text{if } \exists pid{:}\pi_1 a \pi_2 \in \widehat{P}(a) \text{ such that } pid{:}\pi_1 a \pi_2 \sqsubseteq \widehat{\pi} \text{ and } pid{:}\pi_1 \in \widehat{P}(a). \\
&= \widehat{P} \quad \text{if } \widehat{\mathrm{NumProcs}}(\widehat{P}(a)) \geq 2 \\
&= \widehat{P}' \cup \left(\bigcup_{e \in \mathrm{Edge}_G(a)} \mathcal{E}^c_G[\![e]\!]\widehat{\pi}\,\widehat{P}' \right) \quad \text{where } \widehat{P}' = \widehat{P} \cup \{a \mapsto \widehat{P}(a) \cup \{\widehat{\pi}\}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}^c_G[\![(a,\mathbf{ctl},n)]\!]\widehat{\pi}\,\widehat{P} &= \mathcal{N}^c_G[\![n]\!]\widehat{\pi} a\,\widehat{P} \\
\mathcal{E}^c_G[\![(a,\mathbf{spawn},n)]\!]\widehat{\pi}\,\widehat{P} &= \mathcal{N}^c_G[\![n]\!]\widehat{\pi}'\,\widehat{P} \quad \text{where } \widehat{\pi}' = \begin{cases} *{:}\epsilon & \text{if } \widehat{\pi} = *{:}\pi \\ \pi_1\pi_2\bar{a}{:}\epsilon & \text{if } \widehat{\pi} = \pi_1{:}\pi_2 \end{cases} \\
\mathcal{E}^c_G[\![(a,\mathbf{msg},n)]\!]\widehat{\pi}\,\widehat{P} &= \mathcal{N}^c_G[\![n]\!]\widehat{\pi}'\,\widehat{P}\emptyset \quad \text{where } \widehat{\pi}' = \begin{cases} *{:}\epsilon & \text{if } \widehat{\pi} = *{:}\pi \\ a{:}\epsilon & \text{otherwise} \end{cases} \\
\mathcal{E}^c_G[\![(a,\mathbf{wild},n)]\!]\widehat{\pi}\,\widehat{P} &= \mathcal{N}^c_G[\![n]\!]*{:}\epsilon\,\widehat{P}\emptyset
\end{aligned}
$$

Figure 9. Analyzing the CFG G for channel c

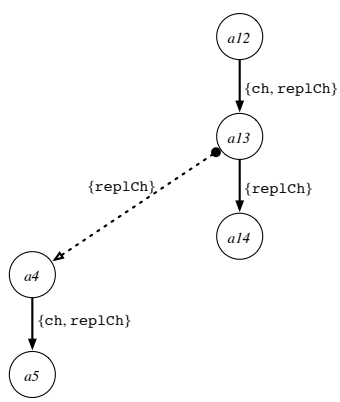


Figure 10. The sub-CFG for `replCh`

results in the following mapping:

$$
\begin{aligned}
\widehat{P_{\texttt{replCh}}}(a_{12}) &= \{\epsilon{:}\epsilon\} \\
\widehat{P_{\texttt{replCh}}}(a_{13}) &= \{\epsilon{:}a_{12}\} \\
\widehat{P_{\texttt{replCh}}}(a_{14}) &= \{\epsilon{:}a_{12}a_{13}\} \\
\widehat{P_{\texttt{replCh}}}(a_4) &= \{a_{12}\bar{a_{13}}{:}\epsilon\} \\
\widehat{P_{\texttt{replCh}}}(a_5) &= \{a_{12}\bar{a_{13}}{:}a_4\}
\end{aligned}
$$

From this map, we see that `replCh` is a one-shot channel.

The analysis for `ch` is more interesting, since it involves spawning, loops, and wild edges. Applying the analysis algorithm to the relevant subgraph produces the following approximation:

$$
\begin{aligned}
\widehat{P_{\texttt{ch}}}(a_2) &= \{\epsilon{:}\epsilon\} \\
\widehat{P_{\texttt{ch}}}(a_3) &= \{\epsilon{:}a_2\} \\
\widehat{P_{\texttt{ch}}}(a_7) &= \{\epsilon{:}a_2a_3\} \\
\widehat{P_{\texttt{ch}}}(a_8) &= \{\pi{:}\epsilon\} \\
\widehat{P_{\texttt{ch}}}(a_4) &= \{\pi{:}a_8,\ \pi{:}a_8a_4a_5a_6\} \\
\widehat{P_{\texttt{ch}}}(a_5) &= \{\pi{:}a_8a_4,\ \pi{:}a_8a_4a_5a_6a_4\} \\
\widehat{P_{\texttt{ch}}}(a_6) &= \{\pi{:}a_8a_4a_5,\ \pi{:}a_8a_4a_5a_6a_4a_5\} \\
\widehat{P_{\texttt{ch}}}(a_9) &= \{\epsilon{:}a_2a_3a_7\} \\
\widehat{P_{\texttt{ch}}}(a_{11}) &= \{*{:}\epsilon\} \\
\widehat{P_{\texttt{ch}}}(a_{12}) &= \{*{:}a_{11}\} \\
\widehat{P_{\texttt{ch}}}(a_{13}) &= \{*{:}a_{11}a_{12}\}
\end{aligned}
$$

where $\pi = a_2a_3\bar{a}_7$. From this map, we see that

$$
\begin{aligned}
\widehat{S_{\texttt{ch}}} &= \{*{:}a_{11}a_{12}\} \\
\widehat{R_{\texttt{ch}}} &= \{\pi{:}a_8,\ \pi{:}a_8a_4a_5a_6\}
\end{aligned}
$$

and thus `ch` is a fan-in channel.

7. Correctness of the analysis

In this section, we show that the static classification of channels from Section 6 correctly follows the dynamic classfication from Section 3.2. In other words, our analysis computes safe approximations of the properties from Section 3.2. The full details of the proofs can be found in the second author's Master's paper [Xia05]; here we cover the ideas underlying the proofs.

First we introduce some notation. We use $\pi^{(i)}$ to denote the i-th program point in π from left, and $\pi^{(-i)}$ to denote the i-th program point in π from right. Let p be a program and c be a channel identifier in p.

To prove our correctness results, we need to instrument our semantics to record the communication history between the dynamic send and receive sites. A communication history H is a subset

$$H \subset \{(\pi_1, k, \pi_2) \mid \pi_1, \pi_2 \in \textsc{CtlPath}, k \in K\}$$

where $(\pi_1, k, \pi_2) \in H$ if there is communication between the dynamic send site π_1 and receive site π_2 on channel instance k in some trace t. For a program p, the initial communication history H will be the empty set. We extend the $\Rightarrow$ relation on traces to also track the communication history. This change only affects the rule

for communication, which becomes

$$\frac{\begin{array}{c} t.\pi_1 = E_1[a_1 : \mathbf{send}(k,\, v)] \text{ is a leaf} \\ t.\pi_2 = E_2[a_2 : \mathbf{recv}\ k] \text{ is a leaf} \end{array}}{\begin{array}{c} H, t \Rightarrow H', t \cup \{\pi_1 a_1 \mapsto E_1[\bullet], \pi_2 a_2 \mapsto E_2[v]\} \\ \text{where } H' = H \cup \{(\pi_1, k, \pi_2)\} \end{array}}$$

For the other evaluation rules, the history does not change; *i.e.*, if $t \Rightarrow t'$ then $H, t \Rightarrow H, t'$. We also extend the definition of the traces of a program:

$$\mathrm{TraceH}(p) = \{H, t \mid \{\}, \{\epsilon \mapsto p\} \Rightarrow^* H, t\}$$

Because our analysis is concerned with only a single channel at a time, we can ignore those parts of the trace where the channel is not live. We formalize this notion in the following definitions.

Definition 1 *For any channel instance k of c in trace $t \in Trace(p)$, the* live projection *of trace t on k, denoted by $t{\downarrow_k}$, is the forest created by removing all the nodes from t in which k does not occur.*

Definition 2 *Given a trace $t \in \mathrm{Trace}(p)$, a channel instance k in t, and a control path π in t, the* live projection *of π on k is defined to be*

$$\pi{\downarrow_k} = \begin{cases} \pi_1 & \text{where } \pi = \pi_2 a \pi_1,\ \pi_1 \in t{\downarrow_k},\ a\pi_1^{(1)} \notin t{\downarrow_k} \\ \pi & \text{otherwise} \end{cases}$$

Because our analysis is designed to modular, given any path π in $t \in Trace(p)$, π may contain program points outside of the module being analyzed. But the following definitions show that for any path in the live projection of a trace, we can approximate the path by collapsing nodes outside the extended CFG into wild edges.

Definition 3 *Let G be an extended CFG for some module in a program p and let π be a path in $t \in Trace(p)$. Then, we say that $\pi \in G$, if for any two adjacent program points $\pi^{(i)}, \pi^{(i+1)}$ in π, there is an edge from $\pi^{(i)}$ to $\pi^{(i+1)}$ in G, and we say that $\pi \pitchfork G$ if there is no $\pi^{(i)}$ in the nodes of G.*

The function $Partition$: CTLPATH $\rightarrow$ CTLPATH* partitions a path π into maximal sub-paths that are either in the module or in the wild.

Definition 4 *Let G be the extended CFG for a module and let π be a path in $t \in Trace(p)$, then*

$$Partition_G(\pi) = \langle \pi_1, \pi_2, ..., \pi_m \rangle$$

where $\pi_1 \pi_2 ... \pi_m = \pi$ and for any $\pi_i \in Partition(\pi)$, π_i is the longest sub-path in π such that $\pi_i \in G$ or $\pi \pitchfork G$.

Definition 5 *Let G be the extended CFG for a module and let π be a path in $t \in Trace(p)$, then for any $\pi_i \in Partition_G(\pi)$,*

$$ApproxPath_G(\pi_i) = \begin{cases} \pi_i & \text{if } \pi_i \in G \\ \epsilon & \text{if } \pi_i \pitchfork G \end{cases}$$

The following lemma asserts that for any path in the live projection of a trace, we can approximate the path by collapsing nodes outside the extended CFG into wild edges.

Lemma 1 *Let $\widehat{G_c}$ be an extended CFG and $k = c@\pi'$ a channel instance in $t \in \mathrm{Trace}(p)$, then for any $\pi \in t{\downarrow_k}$, there exists a $\widehat{\pi} \in \widehat{G_c}$, such that*

$$\widehat{\pi} = ApproxPath_{\widehat{G_c}}(\pi_1) \cdots ApproxPath_{\widehat{G_c}}(\pi_m)$$

where $\langle \pi_1, ..., \pi_m \rangle = Partition_{\widehat{G_c}}(\pi)$

Recall that the approximate paths used to name send/receive sites start from channel creation sites, while the dynamic send/receive sites are labeled by paths starting from the trace root. The next definition and lemma show that for each dynamic send/receive site of any channel instance, there is a corresponding approximate path in the extended CFG, which starts from the channel's creation site.

Given any trace t of program p and channel instance k occurring in t, for any dynamic send/receive sites of k, the following definition of $PathH_{tk}$: CTLPATH $\rightarrow$ CTLPATH* gives us a list of paths that is a flow history and shows how channel instance k flows from its creation site to its send/receive sites. For example, let $\pi \in \mathrm{Sends}_t(k)$ and $PathH_{tk}(\pi) = \langle \pi_1, \pi_2 \rangle$. This means that $\pi_1^{(1)}$ is the creation site of k, k flows through π_1 and then is sent as a message from $\pi_1^{(-1)}$ to $\pi_2^{(1)}$ on some channel instance, and some value is sent on k at $\pi_2^{(-1)}$.

Definition 6 *Given any $H, t \in \mathrm{Trace}(p)$ and $k = c@\pi'$, for any $\pi \in \mathrm{Sends}_t(k) \cup \mathrm{Recvs}_t(k)$, we define*

$$PathH_{tk}(\pi) = \begin{cases} \langle \pi'' \rangle & \text{if } \pi''^{(1)} = \pi'^{(-1)} \\ \langle \pi_1, \pi_2, ..., \pi_m \rangle & \text{otherwise} \end{cases}$$

where $\pi = \pi'''\pi''$, $\pi'' = \pi{\downarrow_k}$, $\pi_m' = \pi'''\pi''^{(1)}$, $\pi_m = \pi''$, $(\pi_i', k_i, \pi_{i+1}') \in H$, $\pi_i = \pi_i'{\downarrow_k}$, and $\pi_1^{(1)} = \pi'^{(-1)}$.

Lemma 2 *Given any $t \in Trace(p)$ and $k = c@\pi'$, let the extended CFG be $\widehat{G_c}$. Then for any $\pi \in \mathrm{Sends}_t(k) \cup \mathrm{Recvs}_t(k)$, $\exists\, \widehat{\pi} \in \widehat{G_c}$, and $\widehat{\pi} = \widehat{\pi_1}\widehat{\pi_2}...\widehat{\pi_m}$, where $PathH_{tk}(\pi) = \langle \pi_1, ..., \pi_m \rangle$.*

This lemma shows that, for any dynamic send/receive site, our analysis computes the approximation of its channel instance flow history in the extended CFG.

The final step is to show that our static classification of channels is *safe* with respect to the dynamic classification, which we do in the following theorems.

Theorem 3 **ONE-SHOT SOUNDNESS**
Let c be a known channel in a module of p. Then, if there exists a trace $t \in Trace(p)$ and instance $k = c@\pi$ in t, such that $|\mathrm{Sends}_t(k)| > 1$, then $\exists \widehat{\pi_1}, \widehat{\pi_2} \in \widehat{S_c}$ such that $\widehat{\pi_1} \neq \widehat{\pi_2}$, or $\widehat{\mathrm{NumProcs}}(\widehat{S_c}) > 1$.

Theorem 3 shows that if there is only one approximate send path, there cannot be more than one dynamic send site for any channel instance k of c. The idea underlying the proof is that two different dynamic send sites have different channel instance flow histories. In the extended CFG, they either have two different approximate paths or have the same approximate path. From Definition 5, we know that if two different flow histories have the same approximation, then the channel instance must escape from the module into the wild. In either case, the number of approximate send paths is more than one. Thus our analysis is sound for the one-shot case.

Theorem 4 **SINGLE-SENDER SOUNDNESS**
If $\exists\, t \in Trace(p)$, $\exists c@\pi$ in t, and $\exists\, \pi_1, \pi_2 \in \mathrm{Sends}_t(c@\pi)$, such that $\mathrm{Proc}(\pi_1) \neq Proc(\pi_2)$, then $\widehat{\mathrm{NumProcs}}(\widehat{S_c}) > 1$.

Theorem 4 shows that if there is only one process in the approximate send set, there cannot be more than one process in the dynamic send sites. The idea underlying the proof is again to consider the channel instance flow history for each dynamic send site. In the channel-instance flow history, the channel instance either stays in the module, escapes into the wild, or is sent as a value to another process. If in any of the above cases the number of processes

involved in the dynamic send sites is more than one, then the approximate paths of flow history have at least two different process ID parts. Thus our analysis is sound for single-sender case.

Theorem 5 SINGLE-RECEIVER SOUNDNESS
If $\exists\, t \in Trace(p)$, $\exists c@\pi$ *in* t, *and* $\exists\, \pi_1, \pi_2 \in \mathrm{Recvs}_t(c@\pi)$, *such that* $Proc(\pi_1) \neq Proc(\pi_2)$, *then* $\widehat{\mathrm{NumProcs}}(\widehat{R_c}) \geq 2$.

Theorem 5 shows that if there is only one process in the approximate receive set, then there cannot be more than one process in the dynamic receive sites. The idea underlying the proof is similar to the one in Theorem 4.

8. Related work

There are a number of papers that describe various program analyses for message-passing languages such as CSP [Hoa78] and CML. We organize our discussion of these analyses by the techniques used.

A number of researchers have used effect-based type systems to analyze the communication behavior of message-passing programs. Nielson and Nielson developed an effects-based analysis for detecting when programs written in a subset of CML have *finite topology* and thus can be mapped onto a finite processor network [NN94]. Debbabi *et al.* developed a type-based control-flow analysis for a CML subset [DFT96], but did not propose any applications for their analysis.

In addition to being used as the basis for analysis algorithms, type systems have been proposed that can be used to specify and verify properties of protocols. For example, Vasconcelos *et al.* have proposed a small message-passing language that uses *session types* to describe the sequence of operations in complex protocols [VRG04]. While this approach is not a program analysis, session types may be a useful way to represent behaviors in an analysis. In particular, they might provide an alternative to our sets of approximate control paths.

There have also been a number of abstract-interpretation-style analyses of concurrent languages that are closer in flavor to the analysis we described in Section 4. Mercouroff designed and implemented an abstract-interpretation style analysis for CSP programs [Mer91] based on an approximation of the number of messages sent between processes. While this analysis is one of the earliest for message-passing programs, it is of limited utility for our purposes, since it is limited to a very static language. Jagannathan and Weeks proposed an analysis for parallel SCHEME programs that distinguishes memory accesses/updates by thread [JW94]. Unfortunately, their analysis is not fine-grained enough for our problem since it collapses multiple threads that have the same spawn point to a single approximate thread. Marinescu and Goldberg have developed a partial evaluation technique for CSP [MG97]. Their algorithm can eliminate redundant synchronization, but, like Mercouroff's work, it is limited to programs with static structure. Martel and Gengler have developed a control-flow analysis that determines an approximation of a CML program's communication topology [MG00]. The analysis uses finite automata to approximate the synchronization behavior of a thread and then extracts the topology from the product automata.

Other researchers have used data-flow techniques to analyze concurrent programs. Some of the earlier work was by Reif, who applied data-flow analysis to an asynchronous CSP-like language [Rei79]. His analysis produced an *event spanning graph*, which is similar to our extended CFG in that it includes edges from send sites to receive sites to track the flow of data between processes. He used these graphs to compute an approximation of reachability for concurrent programs. More recent work by Carlsson *et al.* uses data-flow analysis to determine whether heap-allocated values are sent to other processes (or not) [CSW03]. This information is used to specialize allocations to either the local (per-process) or global heap, which reduces message copying. Their analysis, like ours, starts with 0-CFA to determine the control-flow graph. They then run a first-order data-flow analysis to track the flow of data values from their construction sites to where they are sent as messages.

Colby's abstract-interpretation for a subset of CML is probably the closest to ours [Col95]. His analysis is based on a semantics that uses control paths (*i.e.*, an execution trace) to identify threads. Unlike using spawn points to identify threads (as in [JW94]), control paths distinguish multiple threads created at the same spawn point, which is a necessary condition to understand the topology of a program. The method use to abstract control-paths is left as a "tunable" parameter in his presentation, so it is not immediately obvious how to use his approach to provide the information that we need. His analysis is also a whole-program analysis.

In addition to our prototype implementation of specialized channel operations, there is other evidence that specialized operations are significantly faster than the general-purpose operations. Experience with the existing CML implementation has shown that even in the single-threaded implementation, specialized channel operations can have significant impact on communication overhead. For example, CML provides *I-variables*, which are a form of synchronous memory that supports write-once semantics [ANP89]. Using I-variables in place of channels for one-shot communications can reduce synchronization and communication costs by 35% [Rep99]. In the distributed setting, Demaine describes a protocol for the efficient implementation of a generalized choice construct, where fan-out and fan-in channel operations can be implemented with fewer network messages per user-level communication than many-to-many channel operations [Dem98].

9. Status and future work

The analysis presented in the paper does not include a number of important CML features, such as non-deterministic choice and event combinators. It turns out that to add these features to the analysis is mainly an issue of enriching the CFA to include a representation of approximate event values. We can then further refine our characterization of known channels to distinguish between those that appear in a choice context versus those that do not. The extended CFA analysis also enables other CML optimizations, such as inlining wrapper functions.

We have also considered the question of modelling asynchronous (or buffered) message passing. Our dynamic semantics would have to be extended with a mechanism to track "in-flight" messages, but this extension is not difficult. We believe that our analysis is already correct for buffered channels, since it is not sensitive to the order of messages.

We have prototyped the analysis for a language that is slightly larger than the one in the paper (it has tuples, basic values, conditionals, and a subset of the CML event combinators). The next step will be to extend the analysis to the full set of CML primitives and SML features, such as modules, datatypes, and polymorphism (see [Rep06] for a discussion of the latter). Eventually, we plan to implement the analysis and optimization as a source-to-source tool for optimizing CML modules.

10. Conclusion

We have presented a new analysis technique for analyzing concurrent languages that use message passing, such as CML. Our technique is designed to be applied on individual units of abstraction (*e.g.*, modules). For a given module it determines an approx-

imation of the communication topology for the channels defined in the module. We have shown how this information can be used to replace general-purpose channel operations with more specialized ones. We have also described preliminary results from a prototype implementation of CML primitives in a multiprocessor setting. These results demonstrate that the specialized operations have significantly higher throughput than the general-purpose operations (as much as a factor of 3-4 in some cases).

Our analysis consists of three steps. The first is a new variation of control-flow analysis that we call *type-sensitive* CFA. The type sensitivity of the analysis is what allows us to effectively analyze modules independently of their use. The second step constructs an extended CFG from the CFA results. And the third step analyses the CFG to approximate the numbers of messages and processes involved in communicating through known channels. An important property of this analysis is that it distinguishes between the situation of multiple threads using the same channel and multiple threads using distinct channels. This distinction is key to enabling the specialization of communication primitives. We have also stated and sketched the proof of correctness for our analysis.

We have presented the analysis for a simple concurrent language, but we expect that it will be straightforward to extend to richer languages. The analysis may also be useful for statically detecting other properties of concurrent programs (*e.g.*, deadlock), but we have not explored this direction yet.

References

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, **11**(4), October 1989, pp. 598–632.

[Col95] Colby, C. Analyzing the communication topology of concurrent programs. In *PEPM'95*, June 1995, pp. 202–213.

[CSW03] Carlsson, R., K. Sagonas, and J. Wilhelmsson. Message analysis for concurrent languages. In *SAS '03*, vol. 2694 of *LNCS*, New York, NY, 2003. Springer-Verlag, pp. 73–90.

[Dem97] Demaine, E. D. Higher-order concurrency in Java. In *WoTUG20*, April 1997, pp. 34–47. Available from `http://theory.csail.mit.edu/~edemaine/papers/WoTUG20/`.

[Dem98] Demaine, E. D. Protocols for non-deterministic communication over synchronous channels. In *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'98)*, March 1998, pp. 24–30. Available from `http://theory.csail.mit.edu/~edemaine/papers/IPPS98/`.

[DFT96] Debbabi, M., A. Faour, and N. Tawbi. Efficient type-based control-flow analysis of higher-order concurrent programs. In *Proceedings of the International Workshop on Functional and Logic Programming, IFL'96*, vol. 1268 of *LNCS*, New York, N.Y., September 1996. Springer-Verlag, pp. 247–266.

[FF86] Felleisen, M. and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing (ed.), *Formal Description of Programming Concepts – III*, pp. 193–219. North-Holland, New York, N.Y., 1986.

[FF04] Flatt, M. and R. B. Findler. Kill-safe synchronization abstractions. In *PLDI'04*, June 2004.

[FR99] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *PLDI'99*, May 1999, pp. 37–49.

[GR93] Gansner, E. R. and J. H. Reppy. *A Multi-threaded Higher-order User Interface Toolkit*, vol. 1 of *Software Trends*, pp. 61–80. John Wiley & Sons, 1993.

[Hoa78] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM*, **21**(8), August 1978, pp. 666–677.

[JW94] Jagannathan, S. and S. Weeks. Analyzing stores and references in a parallel symbolic language. In *LFP'94*, New York, NY, June 1994. ACM, pp. 294–305.

[Kna92] Knabe, F. A distributed protocol for channel-based communication with choice. *Technical Report ECRC-92-16*, European Computer-industry Research Center, October 1992.

[Ler00] Leroy, X. *The Objective Caml System (release 3.00)*, April 2000. Available from `http://caml.inria.fr`.

[Mer91] Mercouroff, N. An algorithm for analyzing communicating processes. In *7th International Conference on the Mathematical Foundations of Programming Semantics*, vol. 598 of *LNCS*, New York, NY, March 1991. Springer-Verlag, pp. 312–325.

[MG97] Marinescu, M. and B. Goldberg. Partial-evaluation techniques for concurrent programs. In *PEPM'97*, June 1997, pp. 47–62.

[MG00] Martel, M. and M. Gengler. Communication topology analysis for concurrent programs. In *7th International SPIN Workshop*, vol. 1885 of *LNCS*, New York, NY, September 2000. Springer-Verlag, pp. 265–286.

[MLt] `http://mlton.org/ConcurrentML`.

[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[NN94] Nielson, H. R. and F. Nielson. Higher-order concurrent programs with finite communication topology. In *POPL'94*, January 1994, pp. 84–97.

[Rei79] Reif, J. H. Data flow analysis of communicating processes. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, January 1979. ACM Press, pp. 257–268.

[Rep91] Reppy, J. H. CML: A higher-order concurrent language. In *PLDI'91*, New York, NY, June 1991. ACM, pp. 293–305.

[Rep99] Reppy, J. H. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[Rep06] Reppy, J. Type-sensitive control-flow analysis. In *ML '06*, New York, NY, September 2006. ACM, pp. 74–83.

[Rus01] Russell, G. Events in Haskell, and how to implement them. In *ICFP'01*, September 2001, pp. 157–168.

[Ser95] Serrano, M. Control flow analysis: a functional languages compilation paradigm. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied Computing*, New York, NY, 1995. ACM, pp. 118–122.

[Shi91] Shivers, O. *Control-flow analysis of higher-order languages*. Ph.D. dissertation, School of CS, CMU, Pittsburgh, PA, May 1991.

[VRG04] Vasconcelos, V., A. Ravara, and S. Gay. Session types for functional multithreading. In *CONCUR'04*, vol. 3170 of *LNCS*. Springer-Verlag, New York, NY, September 2004, pp. 497–511.

[Xia05] Xiao, Y. Toward optimization of Concurrent ML. Master's dissertation, University of Chicago, December 2005.

[YYS+01] Young, C., L. YN, T. Szymanski, J. Reppy, R. Pike, G. Narlikar, S. Mullender, and E. Grosse. Protium, an infrastructure for partitioned applications. In *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, January 2001, pp. 41–46.