An Efficient Compilation Framework for Languages Based on a Concurrent Process Calculus

Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa
Department of Information Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 Japan
TEL: +81-3-5800-6913
FAX: +81-3-5689-4365
{oyama,tau,yonezawa}@is.s.u-tokyo.ac.jp

July 1997

Abstract

We propose a framework for compiling programming languages based on concurrent process calculi, in which computation is expressed by a combination of *processes* and *communication channels*. Our framework realizes a *compile-time process scheduling* and *unboxed channels*. The compile-time scheduling enables us to execute multiple independent processes without a scheduling pool operation. Unboxed channels allow us to create a channel without memory allocations and to communicate values on registers. The framework is given as a set of translation rules from a concurrent calculus to an ML-like sequential program. Experimental results show that our compiler can execute sequential programs written in the process calculus only a few times slower than equivalent C programs. This indicates that pure process calculi like ours and programming languages based on them can be implemented efficiently, without losing their simplicity, purity, and elegance.

1 Introduction

In implementing programming languages, we often translate a high level language which has many primitives into an intermediate language which has a smaller set of primitives. Reducing the number of primitives makes it easier to generate code, optimize it, and prove its correctness. Among such intermediate languages are the quadruple in imperative languages, and λ-calculus and Continuation Passing Style (CPS)[1] in functional languages. For concurrent languages, process calculi such as π-calculus[14] and Higher-Order Concurrent Calculus based on Linear Logic (HACL)[13] are good intermediate languages: their syntax and semantics are clear and they have a number of useful theoretical results for optimization[9, 10, 12, 17]. Several concurrent languages are designed and implemented based on process calculi[15, 22, 24].

In this paper, we propose an efficient framework for compiling programming languages which is based on a process calculus. Our target language is a subset of HACL, which is simplified in order to be used as an intermediate language of higher-level languages. In the intermediate language, most constructs[1] are represented by combining processes and channels. For example, in a procedure call, we create a channel, create a process, and perform an inter-process communication via the channel, whether the call is synchronous or asynchronous. Since a large number of processes and channels are created during the execution of a process calculus, the efficiency of the implementation of processes and channels is critical for the overall performance.

[1] More concretely, all constructs except `if` and simple primitives such as arithmetics.

The basic execution model for a process calculus would be as follows. A scheduling pool, which stores all schedulable processes, is maintained during execution. When a process is created dynamically, both the created process and the running process become schedulable. One of the processes is stored in the scheduling pool and the other is executed. When a running process has no natural continuation (e.g., a process failed to receive a value from a channel), one process is extracted from the scheduling pool and executed. Channels are created in heap and all communications between processes are performed through the allocated heap space.

However, this execution model is inefficient in several ways. It frequently manipulates the scheduling pool and a communication always goes through memory. Our execution model described in this paper overcomes the inefficiencies. To reduce the scheduling overhead, we introduce *compile-time scheduling*, which enables us to execute multiple independent processes without a scheduling pool operation. This is achieved by statically keeping track of a set of schedulable processes at each program point. For the communication overhead, we propose *unboxed channel*. In the framework of unboxed channel, a channel is not allocated in the heap, but just put on a register at its creation time. The channel is later elevated to a fully heap-allocated channel as necessary. Under the condition described in this paper, processes can communicate via channels without memory traffic. This unboxed channel framework correctly implements the semantics of first-class channels, while optimizing the performance in common programming idioms.

The structure of this paper is as follows. We explain our language HACL in Section 2, and present a basic execution scheme in Section 3. Section 4 describes the enhanced execution model and Section 5 proposes our compilation algorithm. After showing experimental results in Section 6, we explain related work in Section 7 and finally conclude in Section 8.

2 The Source Language

We define a subset of HACL below.[2]

$$
\begin{array}{lcll}
e\ (\textit{expr}) & ::= & x \mid c \mid op(e_1,\ \ldots,\ e_n) & \\
P\ (\textit{proc}) & ::= & P_1 \,|\, P_2 & \textit{parallel execution} \\
 & & \mid \$x.P & \textit{channel creation} \\
 & & \mid x(y)\texttt{=>}P & \textit{receive} \\
 & & \mid x\texttt{<=}y & \textit{send} \\
 & & \mid \texttt{if } x \texttt{ then } P_1 \texttt{ else } P_2 & \textit{conditional} \\
 & & \mid \texttt{fix } \Gamma \texttt{ in } P & \textit{local process definition} \\
 & & \mid x_0(x_1,\ \ldots,\ x_n) & \textit{process instantiation} \\
 & & \mid \texttt{let } x\texttt{=}e \texttt{ in } P & \textit{let binding} \\
\Gamma\ (\textit{defs}) & ::= & \{f_1(x_1,\ldots,x_n)\ \texttt{=}\ P_1,\ \ldots,\ f_n(x_1,\ldots,x_n)\ \texttt{=}\ P_n\} &
\end{array}
$$

e is an *expression* which is evaluated to a value. An expression is either a variable(x), a constant(c), or a built-in primitive($op(e_1,\ldots,e_n)$). In the following, expressions such as $+(e_1,e_2)$ are abbreviated to e_1+e_2.

P defines *process expressions.* $P_1|P_2$ executes P_1 and P_2 in parallel. $\$x.P$ creates a new channel, binds it to x in P, and executes P. $x(y)\texttt{=>}P$ receives a value from channel x, binds the value to y in P, and executes P. $x\texttt{<=}y$ sends a value y to channel x (and disappears). $\texttt{if } x \texttt{ then } P_1 \texttt{ else } P_2$ executes P_1 if x is true and executes P_2 if x is false. $\texttt{fix } \Gamma \texttt{ in } P$ defines new processes according to Γ and executes P under the new environment. When x_0 is a process defined somewhere, $x_0(x_1,\ldots,x_n)$ executes the body of the process definition, with $x_1,\ldots,x_n$ bound to the corresponding parameters. $\texttt{let } x\texttt{=}e \texttt{ in } P$ binds the evaluation result of e to x and executes P.

Finally, Γ is a set of *process definitions*, each of which is of the form $f(x_1,\ldots,x_n)\ \texttt{=}\ P$, where P is a process expression.

This paper does not deal with the evaluation of expressions, but that of process expressions.

[2]The subset does not have some features found in the original HACL, such as static type, functional primitive, and choice primitive.

2.1 A Sample Program

Below is a HACL program to calculate fibonacci number.

```
fib(r0, x) =
  if (x < 2) then r0<=1
  else $r1.$r2.(fib(r1, x-1) |
               (fib(r2, x-2) |
                r1(v1)=>r2(v2)=>(r0<=v1+v2)))
```

`r0` is a channel to which `fib` will return a calculated value. In the `else` clause, two channels are created by `$r1.$r2.` and two child processes are created asynchronously. In `r1(v1)=>r2(v2)=>(r0<=v1+v2)`, two receive operations from channel `r1` and `r2` are done in turn, and the received values are bound to `v1` and `v2` respectively. Finally, the sum of the values are sent to channel `r0`.

3 A Basic Execution Model

A basic model for executing process expression P would be as follows.

- Initially, P is the only *schedulable* process.
- We maintain a data structure, called *scheduling pool*, in which all schedulable processes are stored.
- To schedule process expression P, one of the following operations is done according to the type of P.

parallel execution (P_1 | P_2) Both P_1 and P_2 become schedulable, thus either P_1 or P_2 is stored in the scheduling pool and the other is executed.

channel creation ($\$x.P$) A new channel is allocated in heap, a pointer to the channel is bound to x, and P is executed.

receive ($x(y)$=>P) x is a pointer to a channel allocated in heap. First, we test whether channel x has a value. If x has one, we extract it, bind it to y, and execute P. Otherwise, P is stored in x, and one schedulable process is selected from the scheduling pool and executed.

send (x<=y) If channel x has a waiting process, we extract the process and execute it, giving the sent value y. Otherwise, we store y in x, select one schedulable process from the scheduling pool, and execute it.

process instantiation ($x_0(x_1, \ldots, x_n)$) We store $x_1, \ldots, x_n$ in locations assigned at compile time and execute the body of x_0.

To sum up, a running process continues its execution as long as it has a natural continuation of the process, putting dynamically-created processes into the scheduling pool. When it has no natural continuations, another schedulable process is picked up from the scheduling pool and executed. Execution model of multithreading languages are basically the same. Concurrent logic programming languages such as KL1[6] and functional languages such as TAM[7] can be described in a similar way, with different details and terminologies.

4 The Enhanced Execution Model

The basic execution model described in the previous section is inefficient in several ways.

Runtime Scheduling Overhead Every time P_1 | P_2 is executed, all of the context in either process (i.e., code address and free variables) are saved in a scheduling pool *at runtime*, though P_1 and P_2 may be scheduled statically and executed in line. We will suffer from the overhead of frequent scheduling pool manipulations.

Communication Overhead All inter-process communications are performed through channel, which is represented as a pointer to a memory location. All the operations to a channel such as putting a value require memory accesses.

We propose the enhanced execution model which overcomes many of the inefficiencies in the basic model. For the scheduling overhead, we propose *compile-time scheduling*. For the communication overhead, we propose *unboxed channels* and further extension to it called *unboxed channel propagation*. The static scheduling enables us to execute multiple independent processes without scheduling pool manipulation. Unboxed channels allows us to create a channel without memory allocation and communicate values on registers. Finally, unboxed channel propagation complements the unboxed channel scheme by further extending the register usage conventions so that unboxed channels can be shared across process instantiation boundaries. As we will show in the following, our enhanced execution model executes procedural abstractions built by a combination of process instantiation and communication via first-class channels as fast as procedure calls in sequential languages.

5 Compilation

5.1 Preliminaries

We express the heart of our compilation method by giving the translation rules from process expressions to an ML-like sequential program which schedules these expressions. After the transformation to a sequential code, register allocation and machine code generation are performed. Specifically, the output ML-like sequential program consists of `let val`, `let fun`, `if-then-else`, sequential composition (`;`), and tail function call. Every function call in the output program is a tail call, and therefore, it is essentially a goto.

5.1.1 Schedulable Closures and the Scheduling Stack

In the output sequential program, all control flows as well as runtime data structures such as the scheduling pool are explicit. The scheduling pool is a list of closures, called *schedulable closures*. It is maintained in LIFO order and is thus hereafter called *the scheduling stack*. In an actual implementation, the scheduling stack may not be a list of closures. It may instead be a contiguous stack as in our implementation. A schedulable closure is executed when it is at the top of the scheduling stack and takes the rest of the scheduling stack as the parameter. The closure eventually pops the next closure from the stack and executes it.

5.1.2 Blocked Closures

When a process tries to receive a value from an empty channel, the continuation which will be executed after the value arrives must be stored in the channel. We actually put a closure to execute the continuation in the channel. We call such a closure *blocked closure* since it represents a computation blocked on a channel. A blocked closure takes two parameters. The first parameter is the scheduling stack and the other is the value finally supplied by a sender. Similar to schedulable closures, they eventually pop the next closure from the stack and execute it.

5.2 The Basic Translation Algorithm

We first present the translation algorithm which realizes the static scheduling and the next section then refines the scheme with unboxed channels. The basic idea of static scheduling is to reduce the frequency of scheduling stack manipulation at runtime. We *statically* identify a set of schedulable processes at each program point, assuming it is a singleton set just after a process is instantiated.

The translation algorithm, denoted by $\mathcal{F}$, takes two parameters and is called in the following form:

$$\mathcal{F}\ U\ k,$$

where U is a set of schedulable processes identified at compile time, and k the name of a variable which, at runtime, refers to the scheduling stack. The invariant is that U and k together constitute all the executable works at that point.

$\mathcal{F}$ outputs a sequential program which eventually pops the next schedulable closure from k and executes it, passing the rest of the closures to it. When U is empty, it simply pops the next schedulable closure and executes it. That is,

$$\mathcal{F}\ []\ k \ = \ \texttt{(hd}\ k\texttt{)}\ \ \texttt{(tl}\ k\texttt{)}^3$$

When U is not empty, on the other hand, $\mathcal{F}$ picks up a process expression from U and schedules it in place. That is,

$$\mathcal{F}\ U\ k \ = \ \mathcal{F}_1\ \ u\ \ U-\{u\}\ \ k,$$

where u is the selected process expression. Notice that the scheduling stack is not manipulated unless U becomes empty.

$\mathcal{F}_1$ is the main algorithm which translates a process expression according to the type of the expression. Figure 1 illustrates the translation algorithm.

In a parallel expression P_1 | P_2, we simply extend U with P_1 and P_2. In a process instantiation $f\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}$, the generated code (1) creates a schedulable closure `s` which schedules U, (2) pushes the closure into the scheduling stack, and (3) calls $f\texttt{((}s\texttt{::}k\texttt{),}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}$, where $\texttt{(}s\texttt{::}k\texttt{)}$ is the extended scheduling stack. (1) and (2) are omitted if U is empty. The parameter list of a process definition is accordingly extended by the pointer to the scheduling stack. The head of the scheduling stack is the 'caller' of the process instantiation.

In a receive expression $r\texttt{(}x\texttt{)=>}P$, the generated code first tests if there is a value in r. If there is one, we extend U by P and a compilation continues with the extended U. Otherwise, the generated code inserts a blocked closure to r and a compilation continues with the original U. In particular, if U is empty (there becomes no statically identified work), it switches to the next closure in the scheduling stack. This is a similar process to procedure return in sequential languages, but does not necessarily imply the computation has finished. It just happens when there becomes no *statically* identified work.

In a send expression $r\texttt{<=}v$, the generated code checks if there is a blocked closure in r and basically schedules it if there is one. An interesting situation occurs if U is not empty *and* a blocked closure is found. We have choices as to which should be scheduled next. Figure 1 illustrates both cases (**Send2a** schedules U first and **Send2b** schedules the blocked closure first) and we currently use **Send2a** to favor the low scheduling overhead.[4]

Heuristics for the Next Process A heuristics must be devised as to which process should be scheduled when $\mathcal{F}$ picks up a schedulable process in U at compile time. Our compiler currently gives a priority to each type of expression and selects one with the highest priority. The priorities, enumerating from higher ones to lower ones, are:

1. parallel expressions
2. send expressions, channel creations, conditionals, or local process definitions
3. receive expressions whose channel is not referenced by any other expressions in U
4. process instantiations
5. receive expressions, the channel of which is referenced by another expression in U

[3]Notational convention here is that typewriter characters express characters which are literally embedded in the resulting ML-like program, whereas lower-case italic characters represent meta variables which are replaced with actual names in the translation. Free meta variables refer to unique names.

[4]The size of the schedulable closure (= the number of free variables) in **Send2a** is always two words (p and v), whereas the that in **Send2b** depends on the size of U and typically larger than two.

Process definition

$\mathcal{G}$ `(`f`(`x_1`, ... ,` x_n`) =` P`)`
  `=` f`(`k`,` x_1`, ... ,` x_n`) =` $\mathcal{F}\ [P]\ k$

Dispatcher

$\mathcal{F}\ U\ k \quad = \quad \mathcal{F}_1\ \ u\ \ U-\{u\}\ \ k$
  where $u \in U$

$\mathcal{F}\ []\ k \quad = \quad$ `(hd` k`) (tl` k`)`

Parallel

$\mathcal{F}_1\ (P_1\ |\ P_2)\ \ U\ k$
  $= \mathcal{F}\ (P_1 :: (P_2 :: U))\ k$

Channel creation

$\mathcal{F}_1$ `($`r`.`P`)` $U\ k\ =$
  `let val` r `= new_channel()`
  `in` $\mathcal{F}\ (P :: U)\ k$ `end`

Process instantiation1

$\mathcal{F}_1$ `(`f`(`x_1`, ...,` x_n`))` $[]\ \ k$
  `=` f`(`k`,`x_1`,...,`x_n`)`

Process instantiation2

$\mathcal{F}_1$ `(`f`(`x_1`, ...,` x_n`))` $U\ \ k\ =$
  `let fun` s`(`k`) =` $\mathcal{F}\ U\ k$
  `in` f`((`s`::`k`),` x_1`,` x_2`, ...,` x_n`) end`

Receive

$\mathcal{F}_1$ `(`r`(`v`)=>`P`)` $U\ k\ =$
  `if (`r *has value*`) then`
    `let val` v `= get_value(`r`)`
    `in` $\mathcal{F}\ (P :: U)\ \ k$ `end`
  `else`
    `let fun` s`(`l`,`v`) =` $\mathcal{F}\ [P]\ \ l$
    `in`
      `put_process(`r`,`s`);`
      $\mathcal{F}\ U\ k$
    `end`

Send1

$\mathcal{F}_1$ `(`r`<=`v`)` $[]\ k\ =$
  `if (`r *has process*`) then`
    `let val` p `= get_process(`r`)`
    `in` p`(`k`,`v`) end`
  `else`
    `put_value(`r`,`v`);`
    $\mathcal{F}\ []\ k$

Send2a

$\mathcal{F}_1$ `(`r`<=`v`)` $U\ k\ =$
  `if (`r *has process*`) then`
    `let val` p `= get_process(`r`)`
      `fun` s`(`k`) =` p`(`k`,`v`)`
    `in` $\mathcal{F}\ U$ `(`s`::`k`) end`
  `else`
    `put_value(`r`,`v`);`
    $\mathcal{F}\ U\ k$

Send2b

$\mathcal{F}_1$ `(`r`<=`v`)` $U\ k\ =$
  `if (`r *has process*`) then`
    `let val` p `= get_process(`r`)`
      `fun` s`(`k`) =` $\mathcal{F}\ U\ k$
    `in` p`((`s`::`k`),`v`) end`
  `else`
    `put_value(`r`,`v`);`
    $\mathcal{F}\ U\ k$

Conditional

$\mathcal{F}_1$ `(if` x `then` P_1 `else` P_2`)` $U\ k\ =$
  `if` x `then` $\mathcal{F}\ (P_1 :: U)\ k$
    `else` $\mathcal{F}\ (P_2 :: U)\ k$

Fix

$\mathcal{F}_1$ `(fix` Γ `in` P`)` $U\ k\ =$
  `let fun` $\mathcal{G}_{map}\ \Gamma$
  `in` $\mathcal{F}\ (P :: U)\ k$ `end`

Figure 1: A basic translation algorithm

By giving the highest priority to parallel expressions, we effectively expand a nested tree of parallel expressions into a flat list of expressions, enhancing the opportunities for the heuristics. Expressions in class 2., except for send expression, never have to create a closure for scheduling purpose, thus receiving the second highest priority.

The priorities for 3., 4., and 5. are more subtle and rely on an assumption on typical programming styles. We assume that channels are likely to be empty when there are unscheduled processes which refer to it. We wish to avoid context switch when we try to receive a value from an empty channel so this suggests giving the lowest priority to such receive expressions. On the other hand, a channel which is *not* referenced to by unscheduled processes tends to already have a value in it. Therefore we give the relatively higher priority to a receive expression from such a channel. We also want to avoid control transfer across process instantiations, thus giving the next lowest priority to process instantiations.

5.2.1 Example

Let us see how the basic translation algorithm schedules the following simple examples:

1. `$r.(r <= x+10 | r(y)=> ...)`

2. given `f(r, x) = r <= x+10`, `$r.(f(r, x) | r(y)=> ...)`

Examples are actually equivalent, if the instantiation of `f` in the second example is substituted by its body (inline expansion). The first one is, omitting untaken branches, converted into:

```
let val r = new_channel()
in if (r has process) then
       (* wake up the process and ... *)
   else
       put_value(r, x+10);
       if (r has value) then
           let val y = get_value(r)
           in ... end
       else
           (* allocate a blocked closure and ... *)
end
```

Moderately clever compilers which keep track of channels just created easily reduce this code to:

```
let val r = new_channel()
in put_value(r, x+10);
   let val y = get_value(r)
   in ... end
end
```

This way, we can further take advantage of the static scheduling.

The second example is translated into the following sequential code.

```
let val r = new_channel()
in let fun s(k) = if (r has value) then
                      let val y = get_value(r)
                      in ... end
                  else
                      (* allocate a blocked closure and ... *)
   in f((s::k), r, x) end
end
```

`k` is the name of a closure at the head of the scheduling stack at this program point.

The definition of `f` is converted to:

```
fun f(k, r, x) = if (r has process) then
                     let val p = get_process(r)
                     in p(x+10, k) end
                 else
                     put_value(r, x+10);
                     (hd k) (tl k)
```

That is, schedulable closure `s` is created and pushed onto the scheduling stack and passed to `f`. `f` sends `x+10` to `r` and return to `s`. `s` will find the value in `r` and continue.

5.2.2 Code Explosion

In the first example in Section 5.2.1, our algorithm generates a code translated from `r(y)=>...` in *both* `then` and `else` clauses of the conditional expression `if (r has process)`. In the `then` clause, our compiler generates a code which (1) wakes up the process, (2) pushes a schedulable

Receive

```
F1^safe (r(v)=>P) U k =
  let fun s(k) = F^safe U k
      fun t(l,v) = F^safe [P] l
  in
    if (r has process) then
      let val v = get_value(r)
      in t((s::k),v) end
    else
      put_process(r,t);
      F [] (s::k)
  end
```

Conditional

```
F1^safe (if x then P1 else P2) U k =
  let fun s(k) = F^safe U k
  in if x then F^safe [P1] (s::k)
          else F^safe [P2] (s::k)
  end
```

Send2a

```
F1^safe (r<=v) U k =
  let fun s(l) = F^safe U l
  in if (r has process) then
        let val p = get_process(r)
            fun t(k) = p(k,v)
        in F [] (s::(t::k)) end
     else
        put_value(r,v);
        F [] (s::k)
  end
```

Send2b

```
F1^safe (r<=v) U k =
  let fun s(k) = F^safe U k
  in
    if (r has process) then
      let val p = get_process(r)
      in p((s::k),v) end
    else
      put_value(r,v);
      F [] (s::k)
  end
```

Figure 2: A summary of the safe translation algorithm

closure to execute the woken process in scheduling stack, and (3) tries to receive a value from a channel `r`. In the `else` clause, it generates a code which (1) sends a value `x+10` to a channel `r` and (2) tries to receive a value from a channel `r`. One process expression `r(v)=>...` is duplicated and translated in two places ((3) in `then` clause and (2) in `else` clause).

As suggested in the above example, our translation scheme incurs an explosive increase of code size because the receive rule duplicates code for U and P, and the send and conditional rules duplicate code for U. We can revert to a more conservative approach in which we share the code for U (or P) by creating a closure which executes U (or P). The translation rules based on the approach is illustrated in Figure 2. They are 'safe' in the sense that no rule duplicates process expressions such as U and P and we are safe from code explosion.

In a receive expression, (1) a schedulable closure for executing U is created and pushed to the scheduling stack, (2) a closure for executing P is created in heap, and (3) we test if a channel r has a value and continue the corresponding execution according to the result. A closure for P must be created in heap because the closure will be put into a channel when a channel has no value. In a conditional and send expression, a closure for executing U is first created. **Conditional** and **Send2b** pushes the closure to the current scheduling stack and a compilation continues with the extended stack ($(s::k)$). On the other hand, **Send2a** first pushes a closure for executing the blocked closure and pushes a closure for U next, when a channel has a blocked closure.

While duplicating some code is crucial for reducing scheduling overhead, some heuristices should be devised which mixes the two strategies. Here we propose one possible mixed strategy which reduces code size, keeping the runtime scheduling overhead small in typical cases. In a typical execution with the scheduling heuristics described above, we rarely try to receive a value from an empty channel. Therefore, generating the codes which are executed only when a block happens without code duplication will not so much sacrifice the overall performance. Based on the observation, our approach duplicates the codes which are still executed without blocks, and it does not duplicate the codes which are executed only when a block happens. A summary of the new

Receive

```
F1 (r(v)=>P) U k =
  if (r has value) then
    let val v = get_value(r)
    in F (P :: U) k end
  else
    let fun s(l,v) = F^safe [P] l
    in
      put_process(r,s);
      F^safe U k
    end
```

Conditional

```
F1 (if x then P1 else P2) U k =
  if x then F (P1 :: U) k
       else F (P2 :: U) k
```

Send2a

```
F1 (r<=v) U k =
  if (r has process) then
    let val p = get_process(r)
        fun s(k) = p(k,v)
    in F^safe U (s::k) end
  else
    put_value(r,v);
    F U k
```

Send2b

```
F1 (r<=v) U k =
  if (r has process) then
    let val p = get_process(r)
        fun s(k) = F^safe U k
    in p((s::k),v) end
  else
    put_value(r,v);
    F U k
```

Figure 3: A summary of the mixed translation algorithm

translation rules to mix the two strategies are shown in Figure 3. Except the underlined parts, the rules are exactly the same as the basic translation rules in Figure 1. The conditional rule is not at all revised but it may be better to generate the code for one of the true and false clauses without duplication, especially when the possibilities of a taken branch are expected to be quite unbalanced.

Indeed the mixed algorithm reduces much code size, our current compiler reduces code size more aggressively. It estimates the code size generated by $\mathcal{F}$ before each translation rule is applied, and it inhibits duplication and uses the safe rules as long as the sum of the estimated code size and the already generated code size exceed a limit given by a programmer. Once the sum falls below the given limit, it thereafter generates a code with the mixed strategy. This approach strongly assures that the compiler never generates too large a code, in exchange for the overheads at some program points where code cannot be duplicated.

In following sample codes, we use a basic translation algorithm because a code generated by the algorithm is easier to read than a code generated by the others.

5.3 Unboxed Channels

5.3.1 The Basic Version

In the first example in Section 5.2.1, the static scheduling generates a code which puts `x+10` to `r` and then receives `x+10` from `r`. As far as this particular example is concerned, we can easily see that there are no need for allocating `r` on heap. We can instead allocate both a flag to tell the state of r (empty, full, ...) and a sent value in r on an one-word register and update the representation of `r` in place.

This scenario of course does not always work. For example, when a reference to a channel is put in a heap-allocated data which may be referenced by unknown processes, we clearly need to allocate heap for it to maintain the correct sharing (aliasing) relationship between processes.

We introduce unboxed channel schemes in two steps. In the basic version, the state of a channel is represented on a register at its creation time and the representation is kept unboxed as long as *it is referenced only by at most one schedulable closure*. It is dynamically elevated to a *boxed* (i.e., heap-allocated) channel when this condition no longer holds.

To realize this unboxed channel scheme, two bit tags are added to all data and it distinguishes four types of data as shown below. The first three are used for unboxed channels and the last one is used for data which are not unboxed channels.

empty_unboxed_channel, in which neither values nor waiting processes are put in the channel.

one_value_unboxed_channel, in which one value is put in the channel, whereas no processes are waiting on it.

one_waitor_unboxed_channel, in which one process is waiting on the channel, whereas no values are put in it.

not_unboxed_channel, which is not an unboxed channel, including an integer, a pointer to a cons cell, and a pointer to a heap-allocated boxed channel.

We represent data itself by remaining bits. For example, on 32 bit machines, we will need to make integers of 30 bits long, so that an integer can be put in an unboxed channel [5].

An unboxed channel is elevated to a boxed one in the following cases:

1. When the channel receives the second value (or waiting process) in the value (or process) queue.
2. When the channel is put into a data structure (such as a cons cell and a channel).
3. When the channel is passed to another process instantiation and the channel is also used in the 'caller' closure.
4. When a blocked closure which references the channel is enqueued in a channel as a waiting process.

In the first case, an unboxed channel is made boxed simply because those values (or processes) cannot be represented in a single word. In the last three cases, the elevation is performed to maintain the correct sharing relationship. Our current compiler simply inserts these runtime checks at all the places where the above operations are performed on a datum which may be an unboxed channel. Actually, the cost of the runtime check is very low in a real program since there are only a few kinds of data which may be an unboxed channel: a channel which is newly created in the same schedulable closure as the operation and a value extracted from the scheduling stack[6].

5.3.2 Unboxed Channel Propagation

Unfortunately, the above basic scheme fails to execute a very frequent idiom efficiently. The frequent idiom is a combination of a channel creation and a parallel expression the left operand of which is a process which receives a value from it, whereas the right operand is a process instantiation which sends a value to it. That is, the frequent idiom looks like:

```
$r.( ... r(v)=> ... | ... f(r, ... ) ... ),
```

where `f` is defined in a process definition. Although the world does not have much experiences on the programming in process calculi, the authors believe that code fragments like the above will become very common, since they correspond to procedure calls found in sequential languages. Notice that in the above idiom, the receiver (left operand) may or may not immediately receive a value from the channel. That is, it represents both synchronous and asynchronous calls.

The basic unboxed channel scheme makes `r` boxed when it is passed to `f`. We extend the basic scheme so that `f` may receive an unboxed channel *in its first parameter* and later *propagate* the (possibly altered) representation of the first parameter to the caller closure. The extended protocol in principle allows a channel to be referenced by *an arbitrary number of schedulable closures* (but

Process definition

$\mathcal{G}\ (f(x_1,\ \ldots,\ x_n)\ \texttt{=}\ P)$

$\quad =\ f(k,\ x_1,\ \ldots,\ x_n)\ \texttt{=}\ \underline{\mathcal{F}\ [P]\ k\ x_1}$

Dispatcher

$\mathcal{F}\ U\ k\ x\ =\ \mathcal{F}_1\ u\ U-\{u\}\ k\ x$

$\quad$ where $u \in U$

$\mathcal{F}\ []\ k\ x\ =\ \texttt{(hd}\ k\texttt{)}\ \underline{\texttt{((tl}\ k\texttt{)},x\texttt{)}}$

Process instantiation

$\mathcal{F}_1\ (f(x_1,\ \ldots,\ x_n))\ U\ k\ x\ =$

$\quad \texttt{let fun}\ \underline{s(k,x_1)\ \texttt{=}\ \mathcal{F}\ U\ k\ x}$

$\quad \texttt{in}\ f((s\texttt{::}k),\ x_1,\ x_2,\ \ldots,\ x_n)\ \texttt{end}$

Receive

$\mathcal{F}_1\ (r(v)\texttt{=>}P)\ U\ k\ x\ =$

$\quad \texttt{if}\ (r\ \textit{has value})\ \texttt{then}$

$\quad\quad \texttt{let val}\ v\ \texttt{= get\_value(}r\texttt{)}$

$\quad\quad \texttt{in}\ \mathcal{F}\ (P :: U)\ k\ x\ \texttt{end}$

$\quad \texttt{else}$

$\quad\quad \texttt{let fun}\ s(l,y,v)\ \texttt{=}\ \underline{\mathcal{F}\ [P]\ l\ y}$

$\quad\quad \texttt{in}$

$\quad\quad\quad \texttt{put\_process(}r,s\texttt{);}$

$\quad\quad\quad \mathcal{F}\ U\ k\ x$

$\quad\quad \texttt{end}$

Send1

$\mathcal{F}_1\ (r\texttt{<=}v)\ []\ k\ x\ =$

$\quad \texttt{if}\ (r\ \textit{has process})\ \texttt{then}$

$\quad\quad \texttt{let val}\ p\ \texttt{= get\_process(}r\texttt{)}$

$\quad\quad \texttt{in}\ p(k,x,v)\ \texttt{end}$

$\quad \texttt{else}$

$\quad\quad \texttt{put\_value(}r,v\texttt{);}$

$\quad\quad \mathcal{F}\ []\ k\ x$

Send2a

$\mathcal{F}_1\ (r\texttt{<=}v)\ U\ k\ x\ =$

$\texttt{if}\ (r\ \textit{has process})\ \texttt{then}$

$\quad \texttt{let val}\ p\ \texttt{= get\_process(}r\texttt{)}$

$\quad\quad \texttt{fun}\ s(k,x)\ \texttt{=}\ \underline{p(k,x,v)}$

$\quad \texttt{in}\ \mathcal{F}\ U\ (s\texttt{::}k)\ x\ \texttt{end}$

$\texttt{else}$

$\quad \texttt{put\_value(}r,v\texttt{);}$

$\quad \mathcal{F}\ U\ k\ x$

Figure 4: A summary of the translation algorithm extended by unboxed channel propagation

not by other types of data including blocked closures). This relaxes the third case in the previous section.

The unboxed channel propagation can be clearly expressed by extending the basic translation rules slightly. A summary of the new translation algorithm is shown in Figure 4. Major differences from the basic version are as follows.

- $\mathcal{F}$ now takes three parameters U, k, and x (see **Dispatcher**). $\mathcal{F}\ U\ k\ x$ returns a sequential code which eventually pop a closure from k and passes the rest of k *and* x to the closure. $\mathcal{F}_1$ now takes four parameters, P, U, k, and x. First three parameters are the same as the basic version. Similar to $\mathcal{F}$, output sequential code generated by $\mathcal{F}_1$ eventually pops the closure from k and returns the rest of k and x to the closure.
- A schedulable closure created at process instantiation $f(x_1,\ \ldots,\ x_n)$ takes an additional parameter which receives the representation of x_1 returned by f (see **Process instantiation** and **Process definition**). f will eventually pops the closure and execute it, giving the representation of x_1 as an argument. The popped closure also eventually pops the next closure (a caller of a caller), passing the representation of x. Because x and x_1 are not always the same, a caller closure must be created even if U is empty.
- A schedulable closure pushed to the scheduling stack at a send expression takes an additional parameter and eventually returns it to the caller, as is (see **Send2a** and **Receive**). This additional parameter is utilized as follows. Suppose the next closure in the scheduling stack expects a value of x when it is executed. When we grow a stack, we must preserve the value of x somehow. Instead of pushing a 'glue' closure which saves x and restores it, we define a protocol in which the new closure takes x and returns it.

[5] In dynamically typed languages like ours, a few more tag bits are used for type tags. For example, our compiler uses more two bits for type tags and integers in our compiler are 28 bits long.

[6] When the basic unboxed channel scheme is extended in the following section, the first parameter of a process instantiation may also be an unboxed channel

5.3.3 Possible Extensions and Improvements

Extension to Unboxed Channel Propagation We have so far assumed that only the first parameter of a process definition can receive unboxed channels. This certainly restricts the programming style which can be efficiently supported. But nothing in the framework prevents us from supporting multiple parameters which can receive unboxed channels. The overhead should be paid, however, to return the representation to the caller. Hence we cannot easily determine how much parameters can hold unboxed channels. There are two possible improvements.

One is achieved by identifying *known process definitions*, which are process definitions whose all call sites are known at compile time. For each call site of such process definitions, we can determine which arguments are actually shared between the newly instantiated process and the current process instantiation. Most aggressively, we can take the union of all such parameters and return those parameters to the caller. The other is to adopt static type system and define a convention in which a parameter is returned to the caller if and only if its type is a channel.

Callee-Save Registers We can reduce more runtime scheduling overheads by callee-save registers, which is a traditional technique in sequential languages. Register usage convention is extended so that callee-save registers can be used in the context of *asynchronous* invocations. Each process definition takes additional parameters and the caller of a process instantiation passes some of its free variables to the process definition. A process returns those parameters to the caller as is. The schedulable closure accordingly takes them as additional parameters. This way we can avoid saving all the free variables at every process instantiation. Expressing callee-save registers this way is first done by [1] and [2]. They describe an effective way of utilizing callee-save registers in sequential programs.

5.3.4 Example

Let us see how the second example in Section 5.2.1 has been improved. First, the definition of `f` is converted in the following sequential code.

```
fun f(k, r, x) = if (r has process) then
                     let val p = get_process(r)
                     in p(k, x+10, r) end
                 else
                     put_value(r, x+10);
                     (hd k) ((tl k), r)
```

The output code for the caller now looks like:

```
let val r = new_channel()
in let fun s(k, r) = if (r has value) then
                         let val y = get_value(r)
                         in ... end
                     else
                         (* allocate a blocked closure and ... *)
   in f((s::k), r, x) end
end
```

First `r` is created as unboxed. After pushing `s` in the scheduling stack `k`, it calls `f`, passing `r` and `x` as arguments. After the invocation, `f` sends a value to `r`. `r` will be empty, hence `f` puts `x+10` by updating `r` in place at `put_value(r, x+10)`.[7] Finally, the updated representation of `r` is passed to the caller `(hd k)`. `s` then obtains `r` and continues to the continuation of the receive expression. We successfully communicate a value without heap allocation. Except the overhead checking whether a channel has a value or not, the sample code is executed as fast as a procedure call in sequential languages.

[7]Updating variables depart from the semantics of ML. Although this can be made explicit by rebinding channels in such places, we prefer to this concise expression. Boxed operations are similarly implicit.

6 Experimental Results

6.1 Source Language

HACL is used as an intermediate language for the concurrent object-oriented language Schematic[22]. Schematic is an extension to Scheme and has some well-known concurrent primitives such as `future` and `touch`, which are originally proposed in Multilisp[8]. Before giving a performance result, we here explain an overview of the surface language Schematic and how some primitives in Schematic are translated to those in HACL.

In Schematic, the expression `(future (`f x`))` asynchronously forks a process f with argument x. The return value of this expression is a channel which is created implicitly to store the returned value from f. The expression `(touch` r`)` tries to receive a value from channel r and returns it if there is one. If there is no value, this expression blocks until a value arrives. The expression `(`f x`)` forks a process f synchronously. The return value of this expression is a value returned by f. Actually, `(`f x`)` is a syntactic sugar for `(touch (future (`f x`)))`. The expression `(make-channel)` returns a newly created channel. The expression `(reply` v r`)` sends a value v to channel r. The value of this expression is undefined. More complicated data structures and primitives such as concurrent object and `pbegin` are implemented by a combination of these primitives.

The primitives in Schematic can be encoded by those in HACL by a straightforward translation. Here we informally describe the encoding. `(future (`f x`))` is translated to `$`r`.(` f`(`r`,` x`) |` P `)`, where P is a process expression translated from a Schematic expression enclosing `(future (`f x`))`. Channel r is implicitly added to f's argument so that f can communicate a value with P. `(touch` r`)` is directly translated to r`(`v`)=>`P, where P is a process expression translated from a Schematic expression enclosing `(touch` c`)`. `(reply` v r`)` is also directly translated to r`<=`v. A synchronous fork expression `(`f x`)`, which is expanded to `(touch (future (`f x`)))`, is translated to `$`r`.(` f`(`r`,` x`) |` r`(`v`)=>`P `)`, where P is a process expression translated from a Schematic expression enclosing `(`f x`)`.

The following code fragment written in Schematic is frequently seen in a typical concurrent programs.

```
(let ((r (future (f x))))
  ...
  ... (touch r) ...
  ...)
```

The above code is translated to the following HACL code, which is very similar to the sample code shown in Section 5.2.1.

```
$r.( f(r, x) | ... r(y)=> ...)
```

The difference between this HACL code and the one translated from `(`f x`)` is whether a receive operation in a right clause is performed immediately or not. As we have explained in Section 5.3.2, our framework can execute this kind of code fragment very efficiently, whether a receive is immediate or not.

The design of Schematic and a translation from Schematic to HACL are out of the main purpose of this paper. Refer to [22] for more details of Schematic.

6.2 Performance Measurement

A Schematic source program is first translated to a HACL program. A HACL program is then translated to an ML-like program based on the algorithm described in this paper. The generated ML-like program is further translated to assembly-like C and compiled to a native code by `gcc`.

We wrote Schematic programs and equivalent C programs for some applications and compared the performance of two programs on SPARCStation20 (HyperSPARC, 150 MHz). In this experiment, runtime type checks in Schematic are omitted for a fair comparison with C.

Benchmark programs are:

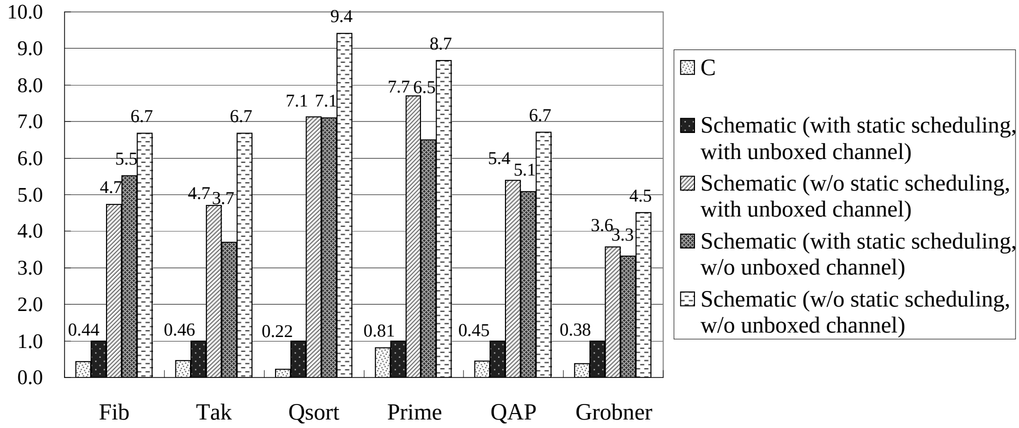

Figure 5: The elapsed times normalized to Schematic with two optimizations (Higher bar indicates slower execution time.)

Fib computes a 30th fibonacci number, 1346269. This program is written in a sequential style, where an answer from one child is immediately received before instantiating another child.

Tak computes a value of Takeuchi function with arguments 24 16 8. This function requires heavily recursive computations as fibonacci function.

Qsort sorts a vector of 10000 integers. The sum of the time to make a vector and sort it ten times is measured. This program destructively updates a vector given as a problem. The dominant computations are a reference and an update to an element of a vector.

Prime computes primes less than 10000 and returns a list of the primes. The sum of the time to calculate the primes ten times is measured. The dominant computations are a division between integers and a cons cell operation.

QAP solves Quadratic Assignment Program, which is a combinatorial optimization problem. It calculates a pseudo optimum solution by tabu search method, which iteratively searches the solution space, utilizing tabu list so as not to turn around the same solutions[11, 16]. The dominant computations of this program are a reference to an element of a vector and a creation of a new vector.

Gröbner computes a Gröbner basis of given polynomials[18]. Polynomials are expressed by combinating cons cells, integers, and symbols. The dominant computations are an integer arithmetic and a cons cell management.

Fib and **Tak** do not allocate any data structures in heap during execution, except closures and channels. **Qsort** also does not allocate any data structures, except closures, channels, and the vector created at the beginning as a problem. All the application programs are written in a sequential style. Heap size in this experiment is one megabytes.

We first show the elapsed times for executing C programs and Schematic programs in Figure 5. The elapsed times are normalized to Schematic with both optimizations and higher bar indicates slower execution time. The elapsed times include garbage collection times. The bar indexed by 'C' shows the elapsed times of C programs. The bars indexed by 'Schematic' show the elapsed times of Schematic programs. We generated four versions of compiled code for each Schematic program, varying the presence of the two optimizations. To generate a code with static scheduling, we used the mixed translation rules in Figure 3. On the other hand, to generate a code without static scheduling, we used the safe translation rules in Figure 2. In the C programs, allocating

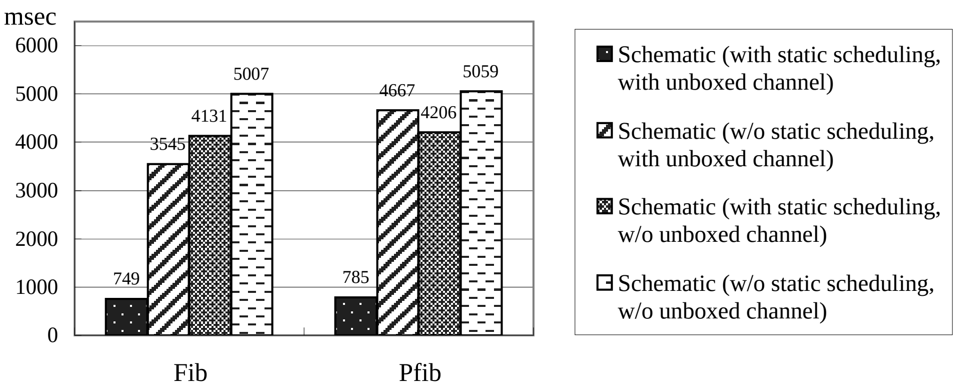

Figure 6: The elapsed times of two Schematic programs (Fib is written in a sequential style and Pfib is written in a parallel style)

data structures such as a cons cell and a vector are performed by `GC_malloc`, which is a function in Boehm's conservative garbage collection library[4, 5]. `GC_malloc` makes a dynamic memory allocation cost small, and liberates us from the burden of inserting a code for freeing memory.

In all the executions of Schematic programs, we never tried to receive a value from an empty channel with the help of the heuristics for process scheduling in the static scheduling algorithm.

In the executions with both two optimizations, no boxed (heap-allocated) channels were created in all the programs because we met none of the four cases in which we must elevate an unboxed channel to a boxed one. Moreover, no heap closure was created in **Fib**, **Tak**, **Qsort**, and **Prime**, except the several heap closures for the processes defined in toplevel. No garbage collection occurred in the four applications. Notice that most of the Schematic programs compiled with both optimizations are executed only a few times slower than the C programs.

When a program is compiled without static scheduling (i.e, without code duplication in our framework), we are subjected to more runtime scheduling overhead. In this compilation, when $r(v)$=>P is executed, a closure for executing P is created in heap before testing whether the channel r has a value, as we described in Section 5.2.2. Therefore, much more heap closures were created in the execution without static scheduling. In addition, unboxed channels which are put into heap-allocated closures are always elevated to a boxed channel so that many boxed channels, which are not seen in the compilation with two optimizations, were created. The elapsed times in Figure 5 clearly tells us that the absence of static scheduling produces a crucial effect on the overall performance.

In the execution without unboxed channel optimization, all channels are created as a boxed channel from the beginning and a channel operation always requires memory manipulations. The absence of unboxed channels increases the cost of channel operations, while the overhead of runtime elevation of a channel and accompanying runtime check listed in Section 5.3.1 is eliminated. In Figure 5, we can easily see that the executions without unboxed channel are much slower than those with unboxed channel. It strongly confirms the usefulness of our unboxed channel scheme.

The speedups achieved by unboxed channel are much more remarkable with static scheduling than without static scheduling. Conversely, the speedups achieved by static scheduling are much more remarkable with unboxed channel. The omission of even one of the two optimizations makes a performance greatly worse. This suggests that a *combination* of static scheduling and unboxed channel optimization brings us an efficient code.

Next, to prove that our scheme is effective in programs written in not only a sequential style but also a parallel style, we wrote a program to calculate a fibonacci number in a parallel style and compared the performance with a sequential one. The parallel program (**Pfib**) has already been presented in Section 2.1. In the execution of **Pfib**, two child processes are instantiated before an answer from one of the children is received. In **Fib**, a second child is instantiated after an answer

	Fib	Pfib	Tak	Qsort	Prime	QAP	Gröbner
basic	11	11	33	91	69	—	—
mixed	11	11	22	58	57	200	1422
safe	8	8	11	32	36	90	382

Table 1: Object code sizes (kilobytes)

		QAP	Gröbner
from Schematic to C	basic	18	—
	mixed	9	34
	safe	7	16
from C to native code	basic	—	—
	mixed	185	1055
	safe	45	170

Table 2: Compilation times (sec)

from a first child is received. Figure 6 shows the elapsed times of two programs. **Pfib** is executed almost as fast as **Fib**, which means that our framework is also powerful for programs written in a parallel style. The ability to execute efficiently a code fragment which encodes an asynchronous call makes our scheme very attractive for implementing a concurrent language. The reason why **Pfib**, compiled without static scheduling and with unboxed channel, is considerably slower than **Fib** is the high ratio of elevated unboxed channels in **Pfib**. The percentage of elevated unboxed channels in all unboxed channels is 50% in **Fib** and 81% in **Pfib**, respectively.

Now, we move a focus to a code size and a compilation time. Table 1 shows the sizes of the generated object codes and Table 2 shows both a compilation time from Schematic to C and a compilation time from C to native code. Compile times for **Fib**, **Pfib**, **Tak**, **QSort**, and **Prime** are omitted because they are very small. We compare the three translation algorithms described in this paper. The first of the three (basic) always duplicates codes, as shown in Figure 1. The second (mixed) duplicates those codes which may be visited even when no block happens and does not duplicate those codes which is visited only when a block happens, as shown in Figure 3. The third (safe) does not duplicate any codes, as shown in Figure 2.

Some blanks in the tables are not filled because Schematic compiler (or C compiler) could not succeed in the compilation because the resulting program is too large. This indicates that the way to avoid code explosion is essential for our scheme to compile realistic programs. From the viewpoints of both code size and compilation time, our mixed compilation algorithm seems to work well. All the compilations by the algorithm are successfully completed and the code sizes are only a few times larger than those of safe algorithm.

7 Related Work

Turner proposed a compilation framework for a concurrent language Pict[15, 24], which is based on π-calculus[14]. In their scheme, a receive (input) expression always allocates a closure, whether the value is present or not, whereas in our scheme, the receive expression immediately continues its body when the value is present. The advantage comes from code duplication. They also propose multiple representation schemes for channels. Their scheme avoids FIFO queue creation in many cases, but, unlike ours, always creates channels in heap.

TAM[7] is an abstract machine for software multithreading. TAM has two-level cost hierarchies for runtime thread scheduling. Low-cost switches can be made between threads which are identified at compile time. This is very similar to our static scheduling scheme. Our current static scheduling realizes zero overhead switches between threads identified at compile time, but this is at the cost of the code size. Further studies are necessary to make a fair comparison.

Reducing the frequency of runtime scheduling operations has also been studied in the context of concurrent logic/functional languages[3, 19]. The basic technique is to find two code fragments that can be statically *merged* (i.e., scheduling one fragment can be safely delayed until the other also becomes schedulable), by dependence analysis. We achieve a similar effect not by dependence analysis but by code duplication. We statically keep track of schedulable code fragments and generate code that schedules both in line, in case both are runnable at runtime.

Kobayashi and Igarashi proposed optimizations for *linear* channels (channels that are used only once) [10, 12]. Their optimizations eliminate a channel creation and indirect communication through the channel entirely, if the channel is used by a receive operation immediately after its creation. The unboxed channel achieves a similar effect by holding the state of an unboxed channel on a register.

The authors' previous work *StackThreads*[21, 23], has proposed the framework of unboxed channels. This work generalized the execution scheme of *StackThreads* and formalized it as a compilation framework in more general settings.

The description of the compilation framework is influenced by Appel's CPS framework for compiling functional languages. Many optimizations they perform can also be incorporated into our compilation framework[1, 2, 20].

8 Conclusion

We proposed an efficient execution scheme for a process calculus and presented a translation algorithm which translates a process calculus into an ML-like sequential program. Our framework has a great generality in that it can compile the program which uses processes and channels in any combination pattern. In our framework, the overhead of a process scheduling has been reduced by *compile-time scheduling*, and that of a communication between processes is reduced by *unboxed channels*. Experimental results showed programs of a process calculus can be executed only a few times slower than equivalent C programs.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Andrew W. Appel and Zhong Shao. Callee-save Registers in Continuation-passing Style. *Lisp and Symbolic Computation 5(3)*, pages 191–221, 1992.

[3] Takuya Araki and Hidehiko Tanaka. Static Granularity Optimization of a Committed-Choice Language Fleng. In *Proceedings of Euro-Par '97 Symbolic Computation*, Passau, Germany, August 1997.

[4] Hans-Juergen Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[5] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[6] T. Chikayama, T. Fujise, and D. Sekita. A Portable and Efficient Implementation of KL1. In *Proceedings of Programming Language Implementation and Logic Programming '94*, volume 844 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, September 1994.

[7] D. E. Culler, S. C. GoldStein, K. E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, pages 347–370, July 1993.

[8] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, April 1985.

[9] Haruo Hosoya, Naoki Kobayashi, and Akinori Yonezawa. Partial Evaluation Scheme for Concurrent Languages and Its Correctness. In *Euro-Par'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 625–632, 1996.

[10] A. Igarashi and N. Kobayashi. Type-Based Analysis of Usage of Communication Channels for Concurrent Programming Languages. Technical report, Department of Information Science, University of Tokyo, 1997.

[11] Hideki Iriyama. Investigation and Improvements of Searching Methods Using Meta Strategies for Quadratic Assignment Problem, 1997. Senior Thesis, University of Tokyo.

[12] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. In *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 358–371, January 1996.

[13] Naoki Kobayashi and Akinori Yonezawa. Higher-Order Concurrent Linear Logic Programming. In *Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes in Computer Science*, pages 137–166. Springer-Verlag, 1995.

[14] Robin Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.

[15] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical report, Computer Science Department, Indiana University, 1997. To appear in Milner *Festschrift*, MIT Press, 1997.

[16] S. Karisch R. E. Burkard and F. Rendl. Quadratic Assignment Problem Library. *European Journal of Operations Research*, 1991. URL of QAPLIB home page `http://www.diku.dk/~Ekarisch/qaplib/`

[17] Davide Sangiorgi. The name discipline of uniform receptiveness. In *Proceedings of 24th International Colloquium on Automata, Languages, and Programming*, July 1997.

[18] Tateaki Sasaki. Some Algebraic Algorithms Based on Head Term Elimination over Polynomial Rings. In J. H. Davenport, editor, *Proceedings of European Conference on Computer Algebra*, volume 378 of *Lecture Notes in Computer Science*, pages 348–354, LeipZig, GDR, June 1987. Springer-Verlag.

[19] Klaus E. Schauser, David E. Culler, and Seth C. Goldstein. Separation Constraint Partitioning — A New Algorithm for Partitioning Non-Strict Programs into Sequential Threads. In *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 259–272, January 1995.

[20] Zhong Shao and Andrew W. Appel. Space-Efficient Closure Representations. In *Proceedings of 1994 ACM Conference on Lisp and Functional Programming*, pages 150–161, June 1994.

[21] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. *StackThreads*: An Abstract Machine for Scheduling Fine-Grain Threads on Stock CPUs. In *Proceedings of Workshop on Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, 1994.

[22] Kenjiro Taura and Akinori Yonezawa. Schematic: A Concurrent Object-Oriented Extension to Scheme. In *Proceedings of Workshop on Object-Based Parallel and Distributed Computation*, volume 1107 of *Lecture Notes in Computer Science*, pages 59–82. Springer-Verlag, 1996.

[23] Kenjiro Taura and Akinori Yonezawa. Fine-grain Multithreading with Minimal Compiler Support—A Cost Effective Approach to Implementing Efficient Multithreading Languages. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[24] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1995.