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

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 CPS [1] in functional languages. For concurrent languages, process calculi such as π-calculus [8] and HACL [7] are good intermediate languages: their syntax and semantics are clear, and they have a number of useful theoretical results for optimization [4–6, 11]. Several concurrent languages are designed and implemented based on process calculi [10, 13, 15].

In this paper, we propose a framework for compiling programming languages which is based on a process calculus. Our target language is a subset of HACL, in which most constructs 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 quite essential for the overall performance.

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), one process is extracted from the scheduling pool and executed. Channels are created in a 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 many of 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*, which allows us to create a channel without memory allocations and to communicate values on registers. In the framework of unboxed channel, a channel is not allocated in a heap, but just put on a register at its creation time. The channel is later elevated to a fully heap-allocated channel as necessary.

The structure of this paper is as follows. We explain our language HACL in Sect. 2 and present our compilation algorithm in Sect. 3. After showing experimental results in Sect. 4, we explain related work in Sect. 5 and finally conclude in Sect. 6.

2 The Source Language

We define a subset of HACL below.[1]

$$
\begin{aligned}
e(expr) &::= x \mid c \mid op(e_1, \cdots, e_n) \\
P(proc) &::= P_1 \mid P_2 \mid \$x.P \mid x(y)\texttt{=>}P \mid x\texttt{<=}y \mid \texttt{if } x \texttt{ then } P_1 \texttt{ else } P_2 \\
&\quad \mid \texttt{fix } \Gamma \texttt{ in } P \mid x_0(x_1, \cdots, x_n) \mid \texttt{let } x = e \texttt{ in } P \\
\Gamma(defs) &::= \{f_1(x_1, \cdots) = P_1, \cdots, f_n(x_1, \cdots) = P_n\}
\end{aligned}
$$

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 ($op(e_1, \cdots, e_n)$).

P is called a *process expression*. $P_1 \mid P_2$ 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 value y to channel x (and disappears). `if` x `then` P_1 `else` P_2 executes P_1 if x is true and P_2 if x is false. `fix` Γ `in` P defines a new process according to Γ and executes P under the new environment. When x_0 is a process defined somewhere, $x_0(x_1, \cdots, x_n)$ executes the body of the process definition, with $x_1, \cdots, x_n$ bound to the corresponding parameters. `let` x `=` e `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 $f(x_1, \ldots, x_n) = P$, where P is a process expression.

[1] The subset does not have some features found in the original HACL, such as static type, function (*lambda* expression), and choice primitive.

3 Compilation

3.1 Preliminaries

We express our compilation method by giving the translation rules from process expressions to an ML-like sequential program which schedules them. Specifically, the output 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.

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*. A closure to be executed is extracted from the top of the scheduling stack and the rest of the stack is passed to it as an argument.

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 computation blocked on a channel. A blocked closure takes two parameters. The first parameter is the scheduling stack. 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.

3.2 The Basic Translation Algorithm

We first present the basic translation algorithm which realizes the static scheduling. The basic idea is to reduce the frequency of runtime scheduling stack manipulations by identifying a set of schedulable processes at each program point.

The algorithm is illustrated in Fig. 1. The translation function $\mathcal{F}$ takes two parameters and is called in the form of $\mathcal{F}\ U\ k$, 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. U and k together constitute all the executable works at that point.

When U is empty, it simply pops the next schedulable closure and executes it. That is, $\mathcal{F}\ []\ k$ = `(hd k) (tl k)` [2]. 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[3]. Notice that the scheduling stack is not manipulated unless U becomes empty.

$\mathcal{F}_1$ is the main function which translates a process expression according to the type of the expression. In a parallel expression $P_1 \mid P_2$, we simply extend U with P_1 and P_2. In a process instantiation $f(x_1, \ldots, x_n)$, the generated code creates a

[2] 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.

[3] See [9] for a heuristics as to which schedulable process $\mathcal{F}$ picks up.

Process definition
$\mathcal{G}\ (f(x_1, \ldots, x_n) = P)$
$= f(k, x_1, \ldots, x_n) = \mathcal{F}\ [P]\ k$

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

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

Process instantiation1
$\mathcal{F}_1\ (f(x_1, \ldots, x_n))\ []\ k$
$= f(k, x_1, \ldots, x_n)$

Process instantiation2
$\mathcal{F}_1\ (f(x_1, \ldots, x_n))\ U\ k =$
let fun $s(k) = \mathcal{F}\ U\ k$
in $f((s::k), x_1, x_2, \ldots, x_n)$ end

Receive
$\mathcal{F}_1\ (r(v)\Rightarrow P)\ U\ k =$
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
$\mathcal{F}_1\ (r\leq v)\ []\ k =$
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
$\mathcal{F}_1\ (r\leq v)\ U\ k =$
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
$\mathcal{F}_1\ (r\leq v)\ U\ k =$
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
$\mathcal{F}_1\ (\text{if } x \text{ then } P_1 \text{ else } P_2)\ U\ k =$
if x then $\mathcal{F}\ (P_1 :: U)\ k$
else $\mathcal{F}\ (P_2 :: U)\ k$

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

Fig. 1. A basic translation algorithm

schedulable closure s which schedules U and calls $f((s::k),x_1,\ldots,x_n)$, where $(s::k)$ is the extended scheduling stack. The head of the scheduling stack, s, is the 'caller' of the process instantiation. In a receive expression $r(x)\Rightarrow P$, 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. In a send expression $r\leq v$, the generated code checks if there is a blocked closure in r and basically schedules it if there is one. If U is not empty *and* a blocked closure is found in r, 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).

Because some translations duplicate a set of process expressions, our scheme

may increase code size exponentially. We can always revert to a more conservative approach to share the code for U by creating a schedulable closure which executes U and pushing it on the scheduling stack. Heuristics should be devised which mix the two strategies [9].

3.3 Unboxed Channels

The Basic Version When both a send operation to a channel and a receive operation from the channel are performed in a schedulable closure and no other operations are performed on the channel, the channel can safely be allocated on a register. However, we cannot always make a channel on a register. For example, when a reference to a channel is put in a heap-allocated data which may be referred to by unknown processes, we clearly need to allocate a heap space for it to maintain the correct sharing (aliasing) relationship between processes.

Based on the above observation, we introduce an unboxed channel scheme as follows. We represent a channel on a register at its creation time and elevate the representation to a *boxed* (i.e. heap-allocated) channel as necessary. The basic version keeps a channel unboxed as long as *it is referenced to only by at most one schedulable closure* (and not from other types of data including blocked closures). It is elevated to a boxed channel when this condition no longer holds.

More concretely, an unboxed channel is elevated to a boxed one in the following cases. When an unboxed channel receives the second value/process in its value/process queue, it is made boxed simply because those values/processes cannot be represented in a single word. The elevation is also performed when the correct sharing relationship cannot be maintained, that is, when an unboxed channel is put in a data structure (such as a cons cell and a channel), when an unboxed channel is passed to another process instantiation, and when a blocked closure which refers to an unboxed channel is enqueued in a channel. Our current compiler inserts runtime checks at all the places where the above operations are performed on the data which may be an unboxed channel.

To implement unboxed channels, two bit tags are added to all the data. The tag distinguishes the four types of data: an empty unboxed channel, an unboxed channel with one value, an unboxed channel with one waiting process, and a data which is not an unboxed channel.

Unboxed Channel Propagation Unfortunately, the above basic scheme fails to execute a very frequent idiom efficiently. The idiom looks like:

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

which corresponds to a procedure call found in sequential languages.

The basic translation algorithm 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 to by *an arbitrary number of schedulable closures* (but not by other types of data including blocked closures).

Process definition

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

Dispatcher

$\mathcal{F}\ U\ k\ x \quad = \quad \mathcal{F}_1\ u\ U-\{u\}\ k\ x$
where $u \in U$
$\mathcal{F}\ []\ k\ x \quad = \quad$ (hd k) $\underline{((\text{tl}\ k), x)}$

Process instantiation

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

Receive

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

Send1

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

Send2a

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

Fig. 2. A summary of the new translation algorithm

A summary of the new translation algorithm is shown in Fig. 2. Major differences from the basic version are underlined in the figure. $\mathcal{F}$ now takes three parameters U, k, and x. $\mathcal{F}\ U\ k\ x$ returns a sequential code which eventually pops the next closure from k and passes the rest of k and x to the closure. Schedulable closures created at process instantiation $f(x_1, \ldots, x_n)$ take an additional parameter which receives the representation of x_1 returned by f. Blocked closures stored in channels take an additional parameter (y) and eventually return it to the caller, as is. Refer to [9] for more details of the new translation algorithm.

4 Experimental Results

Using HACL as an intermediate language, we implemented a compiler for the concurrent object-oriented language Schematic [13]. A Schematic source program is first translated to a HACL program, and 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 some Schematic programs and equivalent C programs for each application and compared the performance of two programs on SPARCStation20 (HyperSPARC, 150 MHz). The result is shown in Table 1.

In this experiment, runtime type checks in Schematic are omitted for a fair comparison with C. With the assistance of the static scheduling algorithm, no blocked closures are made in any of the programs. Moreover, no boxed (heap-allocated) channels are made in any of the programs because we meet none of the cases in which we must elevate an unboxed channel to a boxed one. The

	fib 30	prime 100000	queen 12	tak 24 16 8
Schematic	850	957	866	934
C	327	587	250	364

Table 1. The results of benchmarks (elapsed time in milliseconds)

main remaining overhead is a runtime check of a state of a channel. Refer to [9] for a more detailed performance evaluation of our compiler.

5 Related Work

Turner proposed a compilation framework for a concurrent language Pict [10,15], which is based on π-calculus [8]. 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. We need more study to selectively duplicate code to avoid code size explosion. 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 [3] provides low-cost dynamic scheduling for threads that share the compile-time environments (contexts). Switching between threads that share a context involves only control transfer. We achieve a similar effect by statically keeping track of schedulable code fragments and generating code that schedules all of them in line. Our approach also eliminates unknown jumps that appear in TAM, at the cost of increased 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 [2,12]. 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) [5, 6]. 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 [14] has proposed the framework of unboxed channels. The presented work generalized the 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].

6 Conclusion

We proposed an efficient execution scheme for a process calculus and presented an algorithm which translates a process calculus into an ML-like program. In our framework, the overhead of a runtime process scheduling is reduced by *compile-time scheduling*, and that of a communication between processes is reduced by *unboxed channels*. Experimental results showed the programs of a process calculus can be executed only a few times slower than equivalent C programs.

References

1. A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
2. T. Araki and H. Tanaka. Static Granularity Optimization of a Committed-Choice Language Fleng. In *Proceedings of Euro-Par '97*, Passau, Germany, August 1997.
3. 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.
4. H. Hosoya, N. Kobayashi, and A. Yonezawa. Partial Evaluation Scheme for Concurrent Languages and Its Correctness. In *Euro-Par'96 Parallel Processing*, volume 1123 of *LNCS*, pages 625–632, 1996.
5. 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. (to appear).
6. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. In *Proceedings of POPL '96*, pages 358–371, January 1996.
7. N. Kobayashi and A. Yonezawa. Higher-Order Concurrent Linear Logic Programming. In *Theory and Practice of Parallel Programming*, volume 907 of *LNCS*, pages 137–166. Springer-Verlag, 1995.
8. R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification.* Springer-Verlag, 1993.
9. Y. Oyama, K. Taura, and A. Yonezawa. An Efficient Compilation Framework for Languages Based on Concurrent Process Calculus. Technical report, Department of Information Science, University of Tokyo, 1997. (to appear).
10. B. C. Pierce and D. 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.
11. D. Sangiorgi. The name discipline of uniform receptiveness. In *Proceedings of 24th International Colloquium on Automata, Languages, and Programming*, July 1997.
12. K. E. Schauser, D. E. Culler, and S. C. Goldstein. Separation Constraint Partitioning — A New Algorithm for Partitioning Non-Strict Programs into Sequential Threads. In *Proceedings of POPL '95*, pages 259–272, January 1995.
13. K. Taura and A. Yonezawa. Schematic: A Concurrent Object-Oriented Extension to Scheme. In *Proceedings of Workshop on Object-Based Parallel and Distributed Computation*, volume 1107 of *LNCS*, pages 59–82. Springer-Verlag, 1996.
14. K. Taura and A. Yonezawa. Fine-grain Multithreading with Minimal Compiler Support—A Cost Effective Approach to Implementing Efficient Multithreading Languages. In *Proceedings of PLDI '97*, 1997.
15. D. N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1995.