Compiling concurrent programs for embedded sequential execution

Bill Lin
Electrical and Computer Engineering Department, University of California, San Diego, La Jolla, CA 92093-0407, USA

Abstract

Embedded applications are often more naturally modeled as a set of concurrent tasks, but yet they are often implemented using a single embedded processor. Traditionally, run-time operating systems have been used to simulate concurrency by timesharing the underlying processor and to facilitate inter-process communication amongst the concurrent tasks. However, this run-time approach to multi-tasking and inter-process communication can often introduce significant overhead to execution times and memory requirements, prohibitive in many cases for embedded applications where processor and memory resources are scarce. In this paper, we present compilation techniques that can statically resolve concurrency at compile-time so that the resulting code produced can be sequentially executed on an embedded processor without the need for a run-time scheduler. Our techniques are based on a novel Petri net theoretic framework. In particular, we show how a concurrent program specification can be transformed into an intermediate Petri net representation. We then show how the intermediate Petri net may be statically scheduled to produce a sequential state machine model that can be sequentially executed directly on an embedded processor without a run-time operating system. In practice, this technique produces efficient results. However, theoretically, it is possible for the resulting state machine to become very large, resulting in code explosion. To circumvent this limitation, we describe a compositional approach that can scale well to large applications and is immune to code explosion.

© 2006 Published by Elsevier B.V.

1. Introduction

Software is playing an increasingly important role in embedded systems. This trend is being driven by a wide spectrum of embedded applications, ranging from personal communications systems, to consumer electronics, to automotive, each forming a highly competitive segment of the embedded systems market. In many cases, the software runs on a processor core that is integrated as part of a VLSI chip.

While high-level language compilers exist for implementing sequential programs on embedded processors [1–5], e.g. starting from C, and improved compilers are emerging for digital signal processing (DSP) oriented architectures and in-house application-specific instruction-set processors (ASIPs) [4,5], many embedded software applications are more naturally expressed as concurrent programs, specified in terms of communicating processes. This is because actual system applications are typically composed of multiple tasks. Communicating processes have several attractive properties: they provide a modular way of capturing concurrent behavior, they provide a high-level abstraction for data communication and synchronization, they can be hierarchically composed to form larger systems, and they define a natural level of granularity for partitioning over different distributed hardware–software architectures.

Currently, the most widely deployed solution is to use an embedded operating system to manage the run-time scheduling of processes and inter-process communication. However, this solution can introduce significant overhead to execution times and memory requirements. The execution time overhead is prohibitive in embedded applications where performance is paramount. The memory overhead often translates directly to silicon cost for many system-on-a-chip applications where the program and data memories are partly on-chip. Handcrafted solution is another commonly used approach where concurrent programs are manually rewritten in terms of a sequential program by the designer. This approach is tedious, timing consuming, and
Several alternative high-level approaches have been proposed. Static data-flow solutions [6–8], successfully used to design DSP-oriented systems, achieve compile-time scheduling at the expense of disallowing conditional and non-deterministic execution. Other researchers have considered hybrid approaches [9,10] that generate application-specific run-time schedulers to handle the multi-tasking of conditional and non-deterministic computations. Another important body of work is based on a reactive synchronous specification model [11–13]. These compilation techniques are based on a strong synchrony hypothesis that makes two fundamental assumptions: the existence of a global clock abstraction to discretize computation over instances, and computation conceptually takes no time within each instance. In contrast, our work is based on a model of asynchrony where the concurrent parts can evolve independently and only synchronize where specified.

In this paper, we present new compilation techniques that can generate efficient sequential code from asynchronous process-based program specifications. The resulting sequential code generated can be executed on an embedded processor without the need for a run-time scheduler. The input specification is captured in a C-like programming language that has been extended with mechanisms for concurrency and communication. These extensions are based on the model of Communicating Sequential Processes (CSP), as defined by Hoare [14]. This program specification is described in Section 2. From the input program, an intermediate interpreted Petri net representation is first constructed. This intermediate representation is discussed in Section 3.

A key advantage of this intermediate construction is that the ordering relations across process boundaries are made explicit in the derived Petri net model. Our compilation approach makes use of this partial order information to statically schedule the Petri net to produce a sequential state machine model that can be sequentially executed directly on an embedded processor without a run-time operating system. The sequential state machine produced may be represented as an ordinary C program, which can then be readily retargeted to different processors using processor-specific code generators. Process-level concurrency is statically compiled away while retaining as much partial order information as possible so that maximal freedom is given to the subsequent code generation tools to optimize the scheduling of instructions. This Petri net theoretic compilation method is detailed in Section 4.

In practice, this method produces efficient results. However, theoretically, it is possible for the resulting state machine to become very large, resulting in code explosion. To circumvent this limitation, we describe a compositional approach that first transforms the initial input specification into a set of interacting Petri net components. We then apply the same static scheduling method on each Petri net component to produce a corresponding state machine. The resulting set of interacting state machines are then mapped into a single sequential C program for further processor-specific code generation. In the degenerate case, each process in the initial input specification is mapped into a separate Petri net component. In this case, the size of the resulting sequential code is directly proportional to the size of the original concurrent specification. Thus, this technique can scale well to large applications and is immune to code explosion. This compositional method is detailed in Section 5. The degenerate case is discussed in Section 6.

Finally, in Section 7, we present experimental results to demonstrate the potentials for significant improvements over current run-time solutions.

2. Programming model

In this work, we use a process-based specification as the user-level programming model. Our programming model looks like a C program: the syntactic structure and expression syntax are nearly identical. However, our programming model provides language mechanisms not found in C for specifying processes and channel communications, based on the CSP formalism [14]. In addition to its expressive power to handle parallelism and communication, CSP has a rigorously defined semantics along with a well-defined algebra to reason about the concurrent behavior, which lends well to formal verification. This section presents a brief overview of our programming model by means of examples.

Our programs are hierarchically composed of processes that communicate through synchronizing channels. A simple program is illustrated in Fig. 1. This example is composed of two processes called ping and pong.

```c
1 /* this is a process */
2 ping (input chan(int) a, output chan(int) b)
3 {
4   int x;
5   for(;;) {
6     x = -a; /* receive */
7     if (x < 100) x = 10 - x;
8     else x = 10 + x;
9     b <= x; /* send */
10  }
11 }
12 /* this is another process */
13 pong (input chan(int) c, output chan(int) d)
```

Fig. 1. Process model.
3. Intermediate representation

Our compilation techniques use an interpreted Petri net model as its intermediate representation. In this section, we first provide basic definitions and classification of Petri nets. We then informally, by means of examples, illustrate how an intermediate Petri net representation may be hierarchically constructed from a program of communicating processes.

3.1. Petri nets

Let $G = (P, T, F, m_0)$ be a Petri net [15], where $P$ is a set of places, $T$ is a set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, and $m_0 : P \rightarrow N$ is the initial marking, where $N$ is the set of natural numbers.

The symbols $\bullet t$ and $t \bullet$ define, respectively, the set of input places and the set of output places of transition $t$. Similarly, $\bullet p$ and $p \bullet$ define, respectively, the set of input transitions and the set of output transitions of place $p$.

A place $p$ is called a conflict place if it has more than one output transition. Two transitions, $t_i$ and $t_j$, are said to be in conflict, denoted by $t_i \# t_j$, if and only if $\bullet t_i \cap \bullet t_j \neq \emptyset$.

A state, or marking, $m : P \rightarrow N$, is an assignment of a non-negative integer to each place. $m(p)$ denotes the number of tokens in the place $p$. A transition $t$ can fire at marking $m_1$ if all its input places contain at least one token. The firing of $t$ removes one token from each of its input places and adds a new token to each of its output places, leading to a new marking $m_2$. This firing is denoted by $m_1 \rightarrow m_2$.

Given a Petri net $G$, the reachability set of $G$ is the set of all markings reachable in $G$ from the initial marking $m_0$ via the reflexive transitive closure of the above firing relation. The corresponding graphical representation is called a reachability graph.

A Petri net $G$ is said to be safe if in every reachable marking, there is at most one token in any place. In this case, we can simply represent each marking $m : P \rightarrow \{0, 1\}$ as a binary assignment.

3.2. Classes of Petri nets

A Marked Graph (MG) is a net $G = (P, T, F, m_0)$ such that $\forall p \in P : |\bullet p| = 1 = |p \bullet|$. MGs cannot model conflicts.

A State Machine (SM) is a net $G = (P, T, F, m_0)$ such that $\forall t \in T : |\bullet t| = 1 = |t \bullet|$. SMs cannot model concurrency.

A Free-Choice Net (FC-net) is a net $G = (P, T, F, m_0)$ such that $\forall t_1, t_2 \in T, t_1 \neq t_2 : \bullet t_1 \cap \bullet t_2 \neq \emptyset \Rightarrow |\bullet t_1| = 1 = |\bullet t_2|$, or $\forall p_1, p_2 \in P, p_1 \neq p_2 : p_1 \bullet p_2 \bullet \neq \emptyset \Rightarrow |p_1 \bullet| = 1 = |p_2 \bullet|$. Every MG and SM is a FC-net. For FC-nets, all conflicts can be decided locally.

Let $G'$ be a subset of a net $G$ generated by a non-empty set $X \subseteq P \cup T$. $G'$ is a MG-Component of $G$ if $\bullet t \cup t \subseteq X$ for every $t \in X$, and $G'$ is a strongly connected MG.

Let $G'$ be a subset of a net $G$ generated by a non-empty set $X \subseteq P \cup T$. $G'$ is a SM-Component of $G$ if $\bullet p \cup p \subseteq X$ for every $p \in X$, and $G'$ is a strongly connected SM.

$G$ is said to be covered by a set of MG-Components if every transition of the net belongs to some MG-Component. $G$ is said to be covered by a set of SM-Components if every place of the net belongs to some SM-Component. Hack [16] proved that a live safe FC-net can always be covered by a set of MG-Components or a set of SM-Components.

3.3. Petri net construction

In [17,18], a process algebra was developed for constructing a Petri net model from a program of communication.
processes. Among other operations, the process algebra defines operators for sequential composition, choice composition, recursive composition, and parallel composition. The reader can refer to [17,18] for details. Here, we intuitively illustrate by means of examples how these operators are used to build up the Petri net intermediate representation.

Consider again the example shown in Fig. 1. The derived Petri net models for processes ping and pong are shown in Fig. 2(a) and (b), respectively, along with their initial markings. These Petri nets can be derived by mapping each leaf operation to a primitive transition. Each transition corresponding to a computation action is assigned a separate action label (e.g. b, c, d, and f in Fig. 2). For communication actions, all communication actions along the same channel are assigned the same label (e.g. c1 and c2 in Fig. 2). These primitive transitions can be mapped to a Petri net by iteratively applying the sequential, choice, and recursive composition operators on them.

Concurrent processes can be composed via parallel composition. In parallel composition, communication actions in fact form synchronization points and are joined together at their common transitions. In Petri net theory, parallel composition is essentially a Cartesian product of the two Petri net processes along common labeled actions. This is illustrated in Fig. 2(c).

Observe that once two Petri nets are composed together, all internal communications between the two nets disappear. The actual send and receive operations are eliminated. Instead, they are replaced with simple assignment statements, thus eliminating the communication overhead. Synchronization is represented by explicit partial orderings at the Petri net level. This is a key property since ordering relations across process boundaries are made explicit in the derived Petri net representation. This ordering relations can be used to statically schedule the operations accordingly at compile time, as discussed in Section 4.

4. Sequential code generation

This section describes a static scheduling procedure that works from an intermediate Petri net representation. This section is divided into two parts. We first introduce some basic notions and the concept of an expansion, which corresponds to an acyclic Petri net fragment. We then describe how sequential code can be generated from the expansions.

4.1. Expansions

Before proceeding, we need to introduce several notions.

**Definition 4.1 (Expansion).** An expansion is an acyclic Petri net with the following properties:

- There is one or more places without input transitions.
- There is one or more places without output transitions.
- There are no transitions without at least one input place or one output place.

The places without input transitions are called initial places. The places without output transitions are called cut-off places.

**Definition 4.2 (Maximal expansion).** Let $G$ be a Petri net and let $m$ be a marking of $G$. The maximal expansion of $G$

![Fig. 2. Derived Petri net representations: (a) ping; (b) pong; (c) system = ping || pong.](image-url)
with respect to \( m, E \), is an acyclic Petri net with the following properties:

- The initial places correspond to \( m \).
- The cut-off places correspond to the set of places encountered when a cycle has been reached.
- \( E \) is transitively closed: for each \( t \in E \) or \( p \in E \), all preceding places and transitions reachable from \( m \) are also in \( E \).

\( m \) is referred to as the initial marking.

Intuitively, the maximal expansion of \( G \) with respect to a marking \( m \) corresponds to the largest unrolling of \( G \) from \( m \) before a cycle has been encountered. Consider the example shown in Fig. 3(a). The corresponding maximal expansion with \( m = (p_1, p_2) \) is shown in Fig. 3(b).

**Definition 4.3 (Cut-off markings).** Let \( G \) be a Petri net, and let \( E \) be a maximal expansion of \( G \) with respect to the initial marking \( m \). A marking \( m_c \) is said to be a cut-off marking if it is reachable from \( m \) and no transitions are enabled to fire. The set of cut-off markings is denoted by \( C(E) \).

For the example shown in Fig. 3, there are two possible cut-off markings \( m_{c_1} = (p_1', p_2') \) and \( m_{c_2} = (p_3', p_4') \), shown, respectively, in Fig. 3(c) and (d).

Our compilation procedure works by generating code from a maximal expansion segment \( E \) obtained by using the initial marking \( m_0 \) as the initial marking for the expansion. Then from each cut-off marking \( m_c \in C(E) \), a new maximal expansion segment \( E_i \) is generated using \( m_c \) as the initial marking. This iteration terminates when all cut-off markings have already been visited. The pseudocode for the overall algorithm is shown below.

```plaintext
compile (G, m_0)
{
    R = \{m_0\};
    push (m_0);
    while ((m = pop()) \neq \emptyset) {
        E = maximal-expansion (G, m);
        static-scheduling (E, m);
        foreach \( m_c \in C(E) \) {
            if \( m_c \notin R \) {
                R = R \cup m_c;
                push (m_c);
            }
        }
    }
}
```

The static-scheduling step is applied to each expansion segment to produce the actual code.

In the example shown in Fig. 3, only two expansion segments are needed. From the initial marking \( m = (p_1, p_2) \), the only cut-off markings reachable are \( m_{c_1} = (p_1, p_2) \) and \( m_{c_2} = (p_3, p_4) \). However, from \( m = (p_3, p_4) \), the only cut-off marking reachable is \( m_c = (p_3, p_4) \) itself, as shown in Fig. 4.

However, in the example shown in Fig. 2, only one expansion segment is needed since the only cut-off marking

![Fig. 3. (a) Petri net example; (b) its maximal expansion; (c) a cut-off marking; (d) another cut-off marking.](image)

![Fig. 4. (a) Another maximal expansion; (b) and (c) cut-off marking.](image)
reachable from the initial marking is the initial marking itself (i.e. \( m = (p_1, p_2) \)).

4.2. Properties

The expansion procedure described in Section 4.1 is guaranteed to converge since the number of possible markings in a Petri net is finite. Hence, the number of expansions or iterations is also finite. Typically, very few expansions are required.

For certain classes of Petri nets, the convergence property is even stronger. In the case of a strongly connected live safe MG, the number of expansions is exactly one. This is because in the case of a strongly connected live safe MG, the initial marking \( m_0 \) forms a minimal feedback arc set. The number of tokens along any directed cycle in the MG in the initial marking is exactly one. Thus, according to Definition 4.2, the maximal expansion of a MG \( G \) with respect to its initial marking \( m_0 \) is exactly defined as the acyclic Petri net \( E \) where both the initial places and the cut-off places correspond exactly to the places marked by \( m_0 \). Thus, the set of cut-off markings for \( E \) contains only the initial marking \( m_0 \).

In the case of a strongly connected live safe FC-net \( G \) that can be covered by a set of strongly connected live safe MG components \( G_1, \ldots, G_n \) such that the initial marking \( m_0 \) of \( G \) restricted to \( G_i \) is also a live safe initial marking for the MG component \( G_i \), the number of expansions is also exactly one. The argument follows a similar line as the argument for the MG case. That is, the initial marking \( m_0 \) corresponds to both the initial places and cut-off places if we maximally expand \( G \) with respect to \( m_0 \). Thus, convergence is guaranteed after one expansion since the set of cut-off markings contains only \( m_0 \).

4.3. Static scheduling

We believe that detailed processor-specific optimizations can only be achieved by optimizing code generators that have been highly optimized to a particular processor architecture. This is because modern processors employ very sophisticated pipelining and superscalar execution schemes that differ from processor to processor.

We take an intermediate approach. Our compilation procedure aims to produce, as intermediate output, plain C code that retains a high degree of parallelism so that the subsequent processor-specific code generation step can produce efficient executable machine code for the target processor.

Definition 4.4 (Static scheduling). Let \( E \) be an expansion segment. \( t_i \) is said to precede \( t_j \) in \( E \), denoted as \( t_i \prec t_j \), if there is a directed path from \( t_i \) to \( t_j \). Let \( \pi : T \rightarrow N \), be a schedule function that assigns a non-negative integer \( \pi(t) \in N \) to every \( t \in E \). A schedule is said to be valid iff it satisfies the following condition: \( \forall t_i, t_j \in E, \text{ if } t_i \prec t_j, \text{ then } \pi(t_i) < \pi(t_j) \).

To illustrate this process, consider the expansion segment shown in Fig. 5(a). A valid schedule is shown. Although this static-scheduling step is closely related to the traditional scheduling problem [19,20], we do not yet perform any detailed scheduling of instructions or any detailed resource allocation here. This is deferred to the final code generation step. However, we can make use of similar heuristics in determining a good high-level scheduling. It is not the intention of this paper to discuss in details the different possible scheduling heuristics. The interested reader can refer to [19,20] for a survey of example techniques.

Given a schedule \( \pi \), a state machine fragment \( SM_\pi \) is constructed. In contrast to the traditional scheduling problem, where typically only data-flow blocks are considered, the control-flow graph generation step is much less straightforward. This is because we can have complex concurrent conditionals where the firing of a transition is dependent on the concurrent control flow and must obey Petri net firing rules. Essentially, the control-flow graph generation step is based on a traversal of \( E \) using Petri net firing rules, but we modify the firing rules so that we proceed in accordance to the levels defined by \( \pi \). For example, the schedule shown in Fig. 5(a) will result in the state machine fragment depicted in Fig. 5(b).

In constructing the state machine fragment for the schedule, we distinguish between two types of states: anchor states and non-anchor states. They are defined as follows.

Definition 4.5 (Anchor states). Let \( E \) be an expansion with respect to an initial marking \( m_{C_i} \). \( C(E) \) be the set of cut-off

---

\( ^1 \) Here, we do not distinguish between \( p_i \) and \( p'_i \) because they simply denote different instances of the same place.

---

Fig. 5. (a) A valid static schedule; (b) corresponding state machine (control-flow graph) fragment.
markings, \( \pi : T \rightarrow N \) be the schedule function for \( E \), \( SM_\pi \) be the corresponding state machine fragment induced by \( \pi \), and \( s_1, s_2, \ldots, s_t \) be the corresponding set of states in the state machine fragment \( SM_\pi \). A state \( s_i \) is said to be an anchor state iff \( s_i = m_c \) or \( s_i \in C(E) \). It is said to be a non-anchor state otherwise.

In Fig. 5(b), anchor states are shown pictorially in double ovals, namely states \( p_1p_2 \) and \( p_3p_4 \). Once the overall state machine has been generated, it can be syntactically translated into plain C program for implementation. There are several ways to perform the syntactical mapping. One way is to use a switch-case structure, as shown below.

```c
enum {p1p2, p3p4} state = p1p2;
generate-program ( )
{
    for(;;) {
        switch (state) {
            case p1p2:
                state = p1p2;
                ...
                state = p3p4;
                break;
            case p3p4:
                ...
                state = p3p4;
                break;
        }
    }
}
```

Using this construction, each case label corresponds to an anchor state (cut-off marking), and each case body corresponds to the code generated for the associated expansion segment. Once the overall control-flow graph has been generated, it can be syntactically translated into plain C for implementation. This last code generation step can leverage upon well-studied standard code optimization techniques [3].

### 4.4. Enhanced cut-offs

The control-flow graph generated in Section 4.1 is essentially a reachability graph for the Petri net with a modified firing rule to consider static scheduling. When traversing an expansion segment \( E \), it is possible that certain markings have already been visited when traversing earlier expansion segments. Such previously visited markings can also serve as a cut-off condition.

In particular, suppose when traversing the expansions, we add also the intermediate markings visited during the traversal to the set of reachable states \( R \) in the procedure compile above. Then we can define an enhanced cut-off criterion as follows:

**Definition 4.6 (Enhanced cut-off markings).** Let \( G \) be a Petri net, \( E \) be a maximal expansion of \( G \) with respect to the initial marking \( m \), and \( R \) be a set of markings already visited. A marking \( m_c \) is said to be an enhanced cut-off marking if it is reachable from \( m \), and either \( m \in R \) or no transitions are enabled to fire. The set of enhanced cut-offs is simply denoted as \( C(E) \).

### 4.5. Benefits

The primary benefit of our compilation procedure is the avoidance of overhead introduce by a run-time scheduler. In addition, as can be seen from the previous section, parallelism can be exploited across process boundaries. Another key benefit is the possibility of code optimization across process boundaries. In addition, our compilation procedure produces an ordinary C program that can be retargeted to different processors. For example, the C program below represents a possible solution to the example shown in Fig. 2(c) using our compilation procedure.

```c
enum {p1p2} state = p1p2;
generate-program ( )
{
    int x, y, z = 0;
    for(;;) {
        switch (state) {
            case p1p2:
                x = 10;
                if (x < 10)
                    x = 10 - x;
                else
                    x = 10 + x;
                y = x;
                z = (z + y) % 345;
                state = p1p2;
                break;
        }
    }
}
```

Once generated into this form, well-studied standard code optimization techniques (e.g. constant propagation, dead-code elimination, etc.) can be applied [3]. In this case, the program can be reduced to a program that repeats

\[
  z = (z + 20) \mod 345
\]

after constant propagation.

```c
generate-program ( )
{
    int z = 0;
    for(;;) {
        z = (z + 20) % 345;
    }
}
```

Recall that this example, though simple, was originally specified as two communicating processes. Such
optimizations were not possible directly at the process-level specification.

5. Compositional sequential code generation

The direct static scheduling procedure outlined in Section 4 can be summarized in Fig. 6(a). In this procedure, we compose all the processes in the system directly into a single intermediate Petri net representation. We then use static scheduling to transform the Petri net into a single sequential state machine model SM for implementation. The reason for transforming down to only a single state machine is because we need to eliminate all the run-time preemption and scheduling. The result is that there is only one program running in the system.

The static scheduling procedure is guaranteed to converge since the number of possible markings in a Petri net is finite. Hence, the number of expansions or iterations is also finite. The intermediate Petri net representation is usually comparable in size to the initial process-based specification. However, under some circumstances, the resulting state machine (representing the control flow) may be very large, hence causing the resulting C code to become very large in size as well. This problem can occur because the resulting state machine must represent explicitly the different combinations of states that the processes can be in with respect to the static schedule.

To circumvent this problem, we describe a compositional procedure that first transforms the initial input specification with respect to the static schedule. Different combinations of states that the processes can be in the resulting state machine must represent explicitly. This problem can occur because the resulting state machine must represent explicitly the different combinations of states that the processes can be in with respect to the static schedule.

To circumvent this problem, we describe a compositional procedure that first transforms the initial input specification with n processes, $P_1, P_2, \ldots, P_n$, into a set of k interacting Petri net components, $N_1, N_2, \ldots, N_k$, such that $k \leq n$. In the degenerate case, $k = n$, each process $P_i$ in the initial specification is mapped to a separate Petri net component $N_i$. We then apply our static scheduling method on each Petri net component to produce a corresponding state machine model. We then map the set of interacting state machines into a single C program for implementation. This is achieved by generating a *time-loop* in software that systematically steps through the interacting state machines without the need for a context-switching run-time operating system. This new design flow is depicted in Fig. 6(b). The choice of composition can be decided by the user through compiler directives. The default can be the degenerate case in which each process mapped to a separate Petri net component.

To facilitate communication between the interacting Petri net components, $N_1, N_2, \ldots, N_k$, we first refine the communication channels between the components using a *handshaking protocol*. The handshaking protocol is used to implement data transmission and synchronization between the components. This *handshake expansion* can be done either at the source code or Petri net level. Table 1 shows the code level expansion a possible handshaking protocol. Other handshake protocols may be used with this compositional approach.

Consider again the ping-pong example in Fig. 2. Here, we will consider the degenerate case where each process, $P_1$ and $P_2$, is mapped to its own Petri net component, $N_1$ and $N_2$, respectively. Using handshake expansion at the code level, we obtain the code in Fig. 7. Each channel is represented by a data variable as well as a handshaking variable sync used to control the channel. The sync signal represents if there is data on the channel. sync = 0 means there is no data available, vice versa. To start a receive operation, the receiver first checks if there is data on the channel by examining if sync is high. When the data are available, it will copy the data and reset sync back to low. Similarly, the send operation will first check

![Fig. 6](image.png)  
Fig. 6. (a) Direct approach; (b) compositional approach.

![Fig. 7](image.png)  
Fig. 7. Code level expansion example.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Code level handshake expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive</td>
<td>x = &lt;-a; while(a_sync == 0); x = a_data; a_sync = 0;</td>
</tr>
<tr>
<td>send</td>
<td>a&lt;- = x; while(a_sync == 1); a_data = x; a_sync = 1;</td>
</tr>
</tbody>
</table>

```plaintext
for (;;) { // x = <-a;
    while (a_sync == 0);
    x = a_data;
    a_sync = 0;
    if (x < 0) {
        x = 10 - x;
    } else {
        x = 10 + x;
    }
}
// b<= = x;
while (b_sync == 1);
    b_data = x;
    b_sync = 1;
}
// y = <-b;
while (b_sync == 0);
    y = b_data;
    b_sync = 0;
    z = (z + y) % 345;
}
```
if the channel is empty (available for sending data). It will then copy the data to the channel data variable and set sync to high. Note that only one handshake control variable is needed: the sender sets the control variable and the receiver resets it.

The expanded code shown in Fig. 7 can be translated into corresponding Petri nets shown in Fig. 8. The \( \epsilon \) transitions correspond to dummy transitions. Again, this handshake expansion can be performed directly at the Petri net level. The handshaking protocol is represented as cycles in the Petri net.

After handshake expansion, each Petri net component can be scheduled into a state machine model using the static scheduling procedure. Each state machine model can then be easily translated into C code. The state machine will return control at the end of each anchor state. All the data within each state machine will be kept as global data by using hierarchical renaming schemes. Therefore, there is no context-switching involved when switching between different state machines. We can make a time-loop to simulate these state machines. At each iteration of the time-loop, all the state machines will be advanced to the next anchor state. The pseudo-code for the state machine implementation and the main time-loop is shown in Fig. 9. The explicit state machine representation acts as a built-in preemption scheme. With the help of the time-loop in the main program, the compile-code implementation eliminates the need for run-time preemption, context switching, and scheduling, which are the major performance overheads in a multi-tasking operating system.

It is important to emphasize again that the resulting program is merely an ordinary C program without any specific system calls to any underlying multi-tasking operating system or the need for one. Hence, it is highly portable and only relies on a conventional optimizing C compiler to produce the final implementation.

Since the resulting program is not intended for human processing, it need not necessarily be readable. One useful optimization is to use goto statements instead of function calls. That is, each state machine model is associated with a label. Instead of making function calls in the main time loop, goto statements are used. This can result in slightly faster implementations.

6. Degenerate case

In the degenerate case, each process is mapped to its own Petri net component. Specifically, we consider the degenerate case where each process in the initial specification is a sequential process (i.e. no par statement used internal to a process). For example, this is the case for the ping-pong example presented in Section 2. In this class of degenerate cases, we see that the resulting Petri net component after handshake expansion is actually already a state machine [15] because each transition has only one predecessor place and one successor place. While the corresponding Petri net component is already a state machine, we still need a procedure to determine which states are anchor states and which states are non-anchor states. Although the static scheduling procedure outlined in Section 4 can perform this task, we describe in this section a simple procedure that avoids the need to perform iterative expansions.

Consider the Petri net component shown in Fig. 8(a) after handshake expansion. This corresponds to the process ping described in Section 2. Note that this is a sequential model because there is only one token flowing
through the model. Starting from the initial marking shown, we identify the maximal expansion and the corresponding set of cut-off places, as shown in Fig. 10(a). Instead of an iterative expansion and scheduling procedure, we can simply use the cut-off places directly as the anchor states. This results in the state machine model shown in Fig. 10(b), where the double circles represent the anchor states. In this example, the state \( p_1 \) can be eliminated because its only output transition is a dummy transition \( \varepsilon \). The reduced state machine model is shown in Fig. 10(c). This state machine model can be syntactically mapped to C code using a switch-case structure, as described in Section 4.

It is important to note that the size of the generated state machine (e.g. Fig. 10(b)) is directly proportional to the size of the corresponding Petri net component (e.g. Fig. 8(a)), which in turn is directly proportional to the size of the initial code description of the corresponding process. This means that the size of the resulting C program is also directly proportional to the size of the original specification. Thus, this technique scales well to large applications and avoids code explosion problems.

7. Implementation and results

The compiler techniques presented in this paper has been implemented. The compiler is implemented as a pre-processor that generates plain C [1], which can then be processed by any available optimizing C compiler for a target processor to produce the executable machine code. This results in a highly portable solution. For comparisons, we implemented a multi-tasking approach using a multi-threading library as well. This multi-tasking approach is implemented using a thread library in Solaris on a Sun platform where each process is implemented as a separate thread.

To evaluate the effectiveness of our new approach, we applied it to an example derived from the RC5 encryption algorithm that is widely used for Internet security applications [21]. RC5 is a fast symmetric block cipher that is suitable for hardware or software implementations. It provides a high degree of security, but yet is exceptionally simple. A novel feature of RC5 is the heavy use of data-dependent rotations. Since a full discussion of the RC5 algorithm is beyond the scope of this paper, the interested reader is referred to [21].

The top-level view of the example is shown in Fig. 11. It consists of an encryption–decryption chain. A stream of plaintext is read via the channel \( pt \). Then the RC5 encryption algorithm is applied on it to produce a stream of ciphertext at channel \( ct \). Then the RC5 decryption algorithm is applied to the ciphertext to decode it back to plaintext again, along channel \( dt \).

![Fig. 11. RC5 encryption chain example.](image-url)
Table 2
Comparing results for the RC5 encryption example on a Sun Ultra-2 running Solaris

<table>
<thead>
<tr>
<th>Size</th>
<th>Single</th>
<th>Composition</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5 M</td>
<td>1.1</td>
<td>1.1</td>
<td>8.8</td>
</tr>
<tr>
<td>2 M</td>
<td>2.0</td>
<td>2.2</td>
<td>34.7</td>
</tr>
<tr>
<td>8 M</td>
<td>6.1</td>
<td>6.3</td>
<td>103.5</td>
</tr>
<tr>
<td>32 M</td>
<td>21.7</td>
<td>22.8</td>
<td>554.5</td>
</tr>
<tr>
<td>72 M</td>
<td>48.2</td>
<td>51.4</td>
<td>1241.3</td>
</tr>
<tr>
<td>512 M</td>
<td>335.9</td>
<td>360.1</td>
<td>8871.3</td>
</tr>
<tr>
<td>Rate</td>
<td>1.510 MB/s</td>
<td>1.411 MB/s</td>
<td>0.058 MB/s</td>
</tr>
</tbody>
</table>

We chose this example because it contains data-dependent loops. Table 2 compares the results generated using three methods: the static-scheduling method described in Section 4, the compositional method described in Section 5, and a multi-tasking approach using the Solaris thread library. The columns are labeled Single, Composition, and Threads, respectively. For the composition-based approach, we mapped each process to its own separate Petri net component. That is, we used the degenerate case as our composition strategy. The table compares the execution times of all three approaches on different size input streams.

The first row corresponds to a 0.5 Mbyte input file, the second row corresponds to a 2 Mbyte input file, and so on, with the largest input size of 512 Mbyte. The CPU-times are reported in seconds on a Sun Ultra-2 workstation running Solaris. The row labeled “rate” summarizes the execution of the three solutions in terms of bytes per second. Comparing CPU-times, the static-scheduling approach is comparable to the compositional approach. The Solaris thread-based implementation is slower due to the overhead introduced by multi-tasking and context switching.

8. Conclusion

We described new static compilation techniques for generating efficient implementations of concurrent programs for embedded applications. Our approach differs from previous approaches for asynchronously communicating processes in that it does not require or generate a multi-tasking run-time operating system for execution. Instead, a plain C program is synthesized at compile time that is readily retargetable to different processors. Besides producing a solution that avoids the overheads associated with a run-time operating system, our approach also makes order relations across process boundaries explicit so that partial ordering information can be exploited for optimization. Furthermore, the generated solution is highly portable since it only requires the availability of a host C compiler to support a particular processor. To circumvent potential code explosion problems, we described a compositional method. In the degenerate case, the size of the resulting C program is directly proportional to the size of the original concurrent specification. Thus, this technique scales well to large applications and is immune to code explosion problems.

References


Further reading