Efficient Compilation of Process-Based Concurrent Programs without Run-Time Scheduling

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

Abstract

Currently, run-time operating systems are widely used to implement concurrent embedded applications. This run-time approach to multi-tasking and inter-process communication can 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 a static compilation approach that generates ordinary C programs at compile-time that can be readily retargeted to different processors, without including or generating a run-time scheduler. Our method is based on a novel Petri net theoretic approach.

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. 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 [11, 1, 16, 8, 13], e.g. starting from C, 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, a high-level abstraction for data communication and synchronization, and 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 add 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 hard to debug. The resulting code is often hard to read and maintain, and is usually extremely difficult to modify to accomodate specification changes.

Several alternative high-level approaches have been proposed. Static data-flow solutions [3], 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, 17] that generate application-specific run-time schedulers to handle the multi-tasking of conditional and non-deterministic computations. Reactive approaches, e.g. Esterel [2], rely on a strong synchrony hypothesis that makes two fundamental assumptions: the existence of a global clock abstraction to discretize computation over instances, and computation takes no time within each instance. This hypothesis is difficult to satisfy for distributed implementations and may not match naturally to many applications from a specification standpoint. 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 a new software synthesis approach for implementing asynchronous process-based specifications 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 CSP formalism [10], as introduced 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. This partial order information is used in the software synthesis step to synthesize at compile-time an ordinary C program that can be readily retargeted to different processors, without requiring a run-time kernel. 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 synthesis method is detailed in Section 4. In Section 5, initial results from an encryption example are presented 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 [10]. 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.

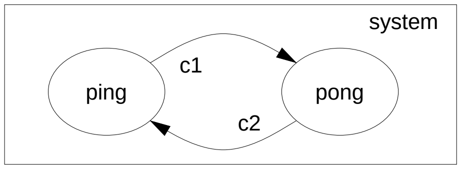


Figure 1: Process model.

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

```
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)
14 {
15   int y, z = 0;
16   for (;;) {
17       d <-= 10;/* send */
18       y = <-c; /* receive */
19       z = (z + y) % 345; /* send */
20   }
21 }

22 /* this is yet another process */
23 system ( )
24 {
25   chan(int) c1, c2;
26   par {
27       ping (c2, c1);
28       pong (c1, c2);
29   }
30 }
```

Channels are declared using the `chan` statement, as exemplified in `Line 2`. The unary receive operator, `<-`, receives data on the channel specified as its right operand. The received value may then be manipulated by other operators, e.g. it can be assigned to a variable, as exemplified in `Line 6`. The send operator, `<-=`, transmits the result of the expression provided as its right operand on the channel specified as its left operand, as exemplified in `Line 9`. Basic control-flow constructs, like `if-then-else`, `for-loops`, and `while-loops`, and basic arithmetic and relational operators, like `+`, `-`, `*`, `%`, and `>`, `>=`, `==`, `!=`, are the same as in C.

There is also an `alt` construct [10], not used here, that provides a mechanism for non-deterministic execution.

Finally, processes can be hierarchically composed to form larger systems, as exemplified by the process `system`. The `par` statement executes the statements in its body in parallel and joins the threads of execution at the end by waiting for all processes to terminate before proceeding. This construct provides a mechanism for invoking concurrency.

3 Intermediate Model

In this section, we first provide basic definitions 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 Basic definitions

Let $G = \langle P, T, F, m_0 \rangle$ be a Petri net [14], 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 $m_0 : P \to N$ 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 number 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 $m_1 \xrightarrow{t} m_2$.

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.

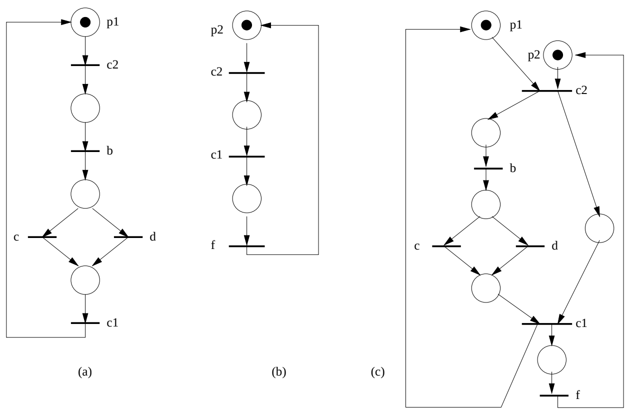
3.2 Intermediate construction

In [5, 19], 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 [5, 19] 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 Fig. 2(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.



	(a)		(b)		(c)
c2:	x=<-a	c2:	d<-=10	c2:	x=10
b:	(x<0)	c1:	y=<-c	b:	(x<0)
c:	x=10-x	f:	z=(z+y)%345	c:	x=10-x
d:	x=10+x			d:	x=10+x
c1:	b<-=x			c1:	y=x;
				f:	z=(z+y)%345

Figure 2: Derived Petri net representations: (a) `ping` (b) `pong` (c) `system = ping` $\|$ `pong`

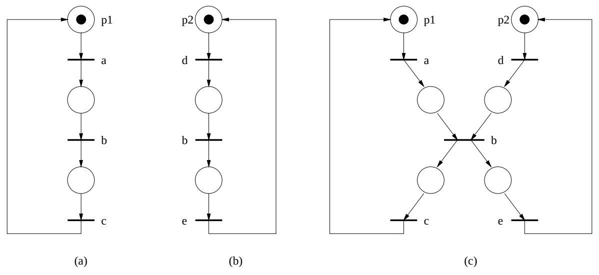


Figure 3: Another example: (a) P (b) Q (c) $P\|Q$

For example, in Fig. 3, two seemingly independent processes can be composed to form a *data-flow-like* model for synthesis, with the previously hidden data dependencies across the processes now made explicit.

4 Static Compilation

This section describes a software synthesis procedure that works from an intermediate Petri net representation. It 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 code can be synthesized 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 are some places, at least one, without input transitions.*
- *There are some places, at least one, 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 with respect to m, E, is an acyclic Petri net with the following properties:*

- *The initial places correspond to the set of places marked by 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. 4(a). The corresponding maximal expansion with $m = \langle p1, p2 \rangle$ is shown in Fig. 4(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 $CM(E)$.*

For the example shown in Fig. 4, there are two possible cut-off markings $m_{c_1} = \langle p1', p2' \rangle$ and $m_{c_2} = \langle p3', p4 \rangle$, shown respectively in Fig. 4(c) and Fig. 4(d).

Our synthesis 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 $m_{c_i} \in CM(E)$, a new

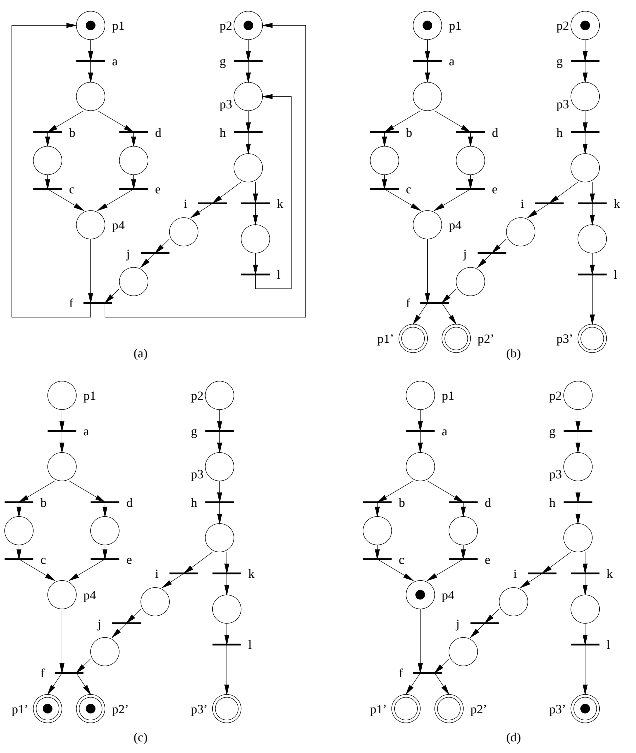


Figure 4: (a) Petri net example. (b) Its maximal expansion. (c) A cut-off marking. (d) Another cut-off marking.

maximal expansion segment E_i is generated using m_{c_i} as the initial marking. This iteration terminates when all cut-off markings have already been visited. This convergence is guaranteed since there are finite number of markings. Typically, very few expansions are required.

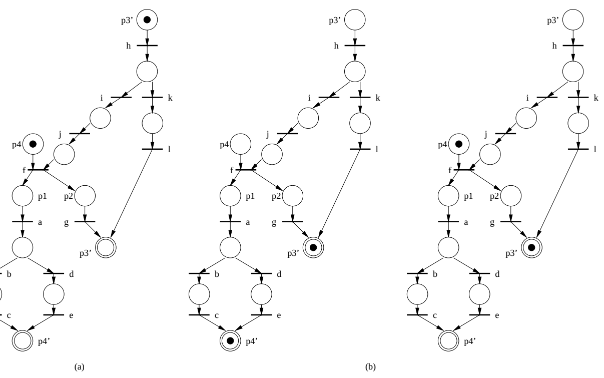


Figure 5: (a) Maximal expansion. (b) Cut-off marking.

Consider the example shown in Fig. 2. In this example, only one expansion segment needs to be considered since the only cut-off marking reachable from the initial marking is the initial marking itself (i.e. $m = \langle p1, p2\rangle$)[1]. For the example shown in Fig. 4, only two expansion segments need to be considered. From the initial marking $m = \langle p1, p2\rangle$, the only cut-off markings reachable are $m_c = \langle p1, p2\rangle$ and $m_c = \langle p3, p4\rangle$. However, from $m = \langle p3, p4\rangle$, the only cut-off marking reachable is $m_c = \langle p3, p4\rangle$ itself, as shown in Fig. 5.

The pseudo-code for the overall algorithm is shown below.

```
soft-synt ($G$, $m_0$)
{
  $EM$ = $\{m_0\}$;
  push ($m_0$);
  while (($m$ = pop()) $\neq \emptyset$) {
     $E$ = maximal-expansion ($G$, $m$);
     code-synthesis ($E$, $m$);
     foreach $m_c \in CM(E)$ {
        if $m_c \notin EM$ {
           $EM$ = $EM \cup m_c$;
           push ($m_c$);
        }
     }
  }
}
```

The `code-synthesis` step is applied to each expansion segment to produce the actual code.

4.2 Expansion-based code generation

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 most modern processors employ very sophisticated pipelining and superscalar execution schemes that differ from processor to processor. We take an intermediate approach. Our software synthesis method 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.

Give an expansion segment E, represented as an acyclic Petri net fragment, our software synthesis method performs a *pre-ordering* of the operations in that segment. During pre-ordering, a *level* assigned to every operation in E. More formally, a pre-ordering is defined as follows:

Definition 4.4 (Pre-ordering) *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 $\pi : T \to N$, be a* pre-ordering function *that assigns a non-negative integer $\pi(t) \in N$ to every $t \in E$. A pre-ordering is said to be* valid *iff it satisfies the following condition: $\forall t_i, t_j \in E$, if $t_i \prec t_j$, then $\pi(t_i) < \pi(t_j)$.*

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

To illustrate this process, consider the expansion segment shown in Fig. 6(a) (corresponding to the example depicted in Fig. 6). Two valid pre-orderings are shown in Fig. 6(b) and Fig. 6(c). Although this pre-ordering step is closely related to the traditional scheduling problem [4, 6], we do not yet perform any detailed scheduling of instructions or any detailed resource allocation here. That is deferred to the final code generation step. However, we can make use of similar heuristics in determining a good pre-ordering. It is not the intention of this paper to discuss in details the different possible scheduling heuristics. The interested reader can refer to [4, 6] for a survey of example techniques.

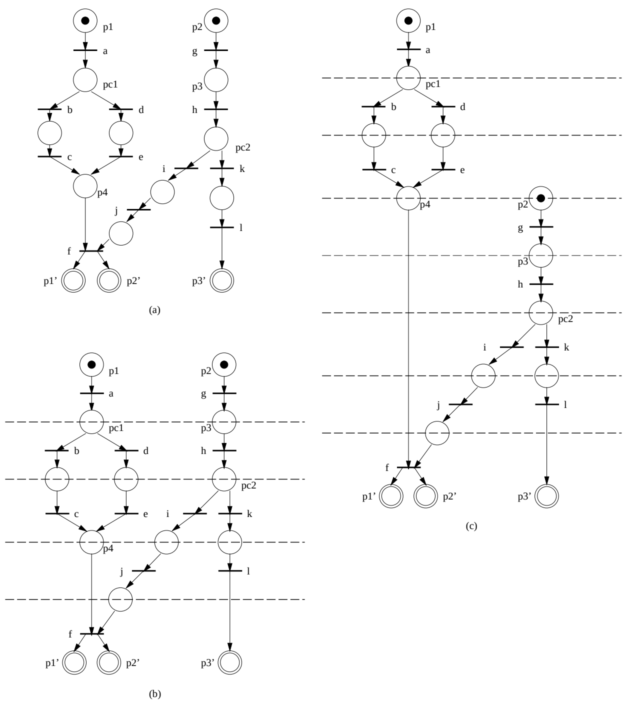


Figure 6: (a) An expansion segment. (b) A valid pre-ordering. (c) Another valid pre-ordering.

Given a pre-ordering π, a control-flow-graph fragment is constructed. In contrast to the traditional scheduling problem, where typically only *data-flow blocks* are considered, the control-flow-graph mapping 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 conflow flow and must obey Petri net firing rules. Essentially, the control-flow-graph generation step is based on a traveral of E, but we modify the Petri net firing rules so that we proceed in accordance to the levels defined by π. For example, the pre-ordering shown in Fig. 6(b) will result in the control-flow-graph fragment depicted in Fig. 7(a). Similarly, Fig. 7(b) shows the resulting control-flow-graph for the pre-ordering shown in Fig. 6(c).

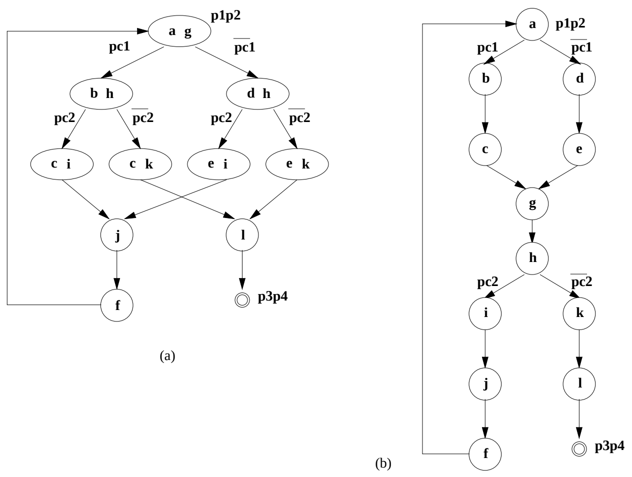


Figure 7: (a) An control-flow-graph fragment. (b) Another control-flow-graph fragment.

4.3 Enhanced cut-offs

The control-flow-graph generated in Section 4.2 is essentially a *reachability graph* for the Petri net with a modified firing rule to consider the pre-orderings. 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 EM in the procedure `soft-synt` above. Then we can define an *enhanced cut-off criterion* as follows:

Definition 4.5 (Enhanecd cut-off markings) *Let G be a Petri net, E be a maximal expansion of G with respect to the initial marking m, and EM 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 $m \in EM$ or no transitions are enabled to fire. The set of enhanced cut-off markings is simply denoted as $CM(E)$.*

4.4 Benefits

The primary benefit of our synthesis approach is the avoidance of overhead introduce by run-time multi-tasking solutions. In addition, parallelism can be exploited across processor boundaries. Another key benefit is the possiblity of code optimization across process boundaries. For example, the C program below represents a possible solution to the example shown in Fig. 2(c) using our synthesis procedure.

```
generated-program ( )
{
    int x, y, z = 0;
p1p2:
    x = 10;
    if (x < 10)
       x = 10 - x;
    else
       x = 10 + x;
    y = x;
    z = (z + y) % 345;
    goto p1p2;
}
```

Once synthesized into this form, well-studied standard code optimization techniques can be applied [1, 16]. In this case, the program can be reduced to a program that repeats `z=(z+20)%345` after constant propagation.

```
generated-program ( )
{
    int z = 0;
p1p2:
    z = (z + 20) % 345;
    goto p1p2;
}
```

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 Example

The synthesis method presented in this paper has been implemented. The compiler is implemented as a preprocessor that generates plain C, 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 multithreading 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 [15]. 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 [15].

The top-level view of the example is shown in Fig. 8. It consists of two encryption-decryption chains. Each chain reads a stream of *plaintext* via channel pt0 (pt1), applies the RC5 encryption algorithm on it to produce a stream of

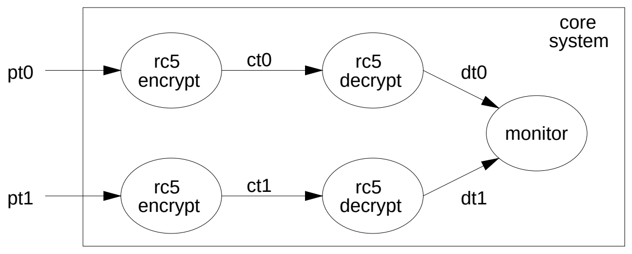


Figure 8: RC5 encryption chain example.

ciphertext at channel ct0 (ct1), then applies the RC5 decryption algorithm to decode the ciphertext back to plaintext again, along channel dt0 (dt1). A monitor process is introduced to merge the two deciphered streams for output.

size	threads	synthesis
40K	2.6	0.04
400K	26.0	0.33
4M	256.6	3.30
rate	15.4KB/s	1.21MB/s
code	87.08KB	8.49KB

Table 1: Comparing results for the RC5 encryption example on a Sun Ultra-1 running Solaris.

We chose this example because it contains both data-dependent loops as well as non-deterministic choices. Table 1 compares the results of our method with a multi-threading library approach. The first part of the table compares the execution times of both approaches on different size input streams. The first row corresponds to a 40K bytes input file, the second row corresponds to a 400K byte input file, and the third row corresponds to a 4M byte input file. The CPU-times are reported in seconds on a Sun Ultra-1 workstation running Solaris. The row labeled "rate" summarizes the execution of the two solutions in terms of bytes per second. Comparing CPU-times, the Solaris thread based implementation is significantly slower than our software code synthesis approach, due to the significant overhead introduced by Solaris threads. The inclusion of Solaris threads also introduced significant overhead in code size. The last row of the table labeled "code" shows the program size of each method.

6 Conclusion

We described a new software synthesis approach towards 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 synthesis algorithms can exploit the partial ordering information for optimization. Furthermore, the synthesized solution is highly portable since it only requires the availability of a host C compiler to support a particular processor.

References

[1] A. V. Aho et al. *Compilers - principles, techniques, and tools*, Reading: Addison-Wesley, 1986.

[2] G. Berry et al. "The synchronous approach to reactive and real-time systems", *IEEE Proceedings*, 1991.

[3] J. T. Buck et al. "Ptolemy: A framework for simulating and prototyping heterogeneous systems", *International Journal on Computer Simulation*, January 1994.

[4] R. Camposano and W. Wolf (editors), *Trends in High-Level Synthesis*, Kluwer Academic Publishers, 1993.

[5] G. de Jong, B. Lin. "A communicating Petri net model for the design of concurrent asynchronous modules", *ACM/IEEE Design Automation Conference*, 1994.

[6] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. Van Meerbergen, S. Note, J.A. Huisken, "Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms", *Proceedings of IEEE*, vol.72, no.2, pp.319-335, February 1990.

[7] P. Godefroid, P. Wolper. "Using partial orders for the efficient verification of deadlock freedom and safety properties", *Lecture Notes in Computer Science*, 575(10):332-342, July 1991.

[8] G. Goossens et al. "Embedded Software in Real-Time Signal Processing Systems: Design Technologies", *Proceedings of the IEEE*, special issue on Hardware/Software Co-Design, 1997.

[9] R. K. Gupta. "Hardware-software cosynthesis of microcontrollers", *Proc. Codes/CASHE*, 1996.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[11] B. W. Kernighan, D. M. Ritchie. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[12] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw Hill, 1978.

[13] P. G. Paulin et al. "Trends in embedded system technology: an industrial perspective", *Hardware/Software Co-Design*, G. De Micheli, M. Sami, Editors, Boston : Kluwer Academic Publishers, 1996.

[14] J.L. Peterson. *Petri net Theory and Modeling of Systems*, Prentice Hall, 1981.

[15] R. L. Rivest. "The RC5 Encryption Algorithm", *Proceedings of the 1994 Leuven Workshop on Algorithms*, Springer-Verslag, 1994.

[16] R. M. Stallman, *Using and porting GNU CC*, Free Software Foundation, June 1993.

[17] F. Thoen et al. "Real-time multi-tasking in software synthesis for information processing systems", *Proc. of ISSS'95*, 1995.

[18] A. Valmari. "A stubborn attack on state explosion", *Proc. of 2nd Workshop on Computer-Aided Verification*, pages 156-165, June 1990.

[19] S. Vercauteren et al. "Derivation of formal representations from process-based specification and implementation models", *Proc. of ISSS'97*, September 1997.