Modular Static Scheduling of Synchronous Data-flow Networks

Marc Pouzet
LRI, Univ. Paris-Sud and IUF
INRIA/Orsay

Pascal Raymond
Verimag-CNRS
Grenoble

Journée du GDR Programmation, 21 octobre 2009
The problem

- **Input**: a *parallel* data-flow network made of synchronous operators. E.g., **LUSTRE, SCADE, SIMULINK**

- **Output**: a sequential procedure (e.g., C, Java) to compute one step of the network: *static scheduling*

Examples: (**SCADE** and **SIMULINK**)
Abstract Data-flow Network and Scheduling

Whatever be the language, a data-flow network is made of:

- **instantaneous** nodes which need their current input to produce their current output. E.g., combinatorial operators.
  
  ↦ atomic *actions*, (partially) ordered by data-dependency

- **delay** nodes whose output depend on the previous value of their input. E.g., \( \text{pre} \) of SCADE, \( 1/z \) and integrators in SIMULINK, etc.
  
  ↦ state variables + 2 side-effect actions read (*set*) and update (*get*)

  ↦ reverse dependency (and allow feed back)

\[ \begin{array}{c}
\text{implemented by} \\
D \\
\hspace{1cm} o
\end{array} \]

\[ \begin{array}{c}
i \\
\downarrow \\
\text{set} \\
\downarrow \\
get \\
\downarrow \\
\hspace{1cm} o
\end{array} \]
Sequential Code Generation

Build a static schedule from a partial ordered set of actions

Code Generation for Synchronous Block-diagram
Sequential Code Generation

Build a static schedule from a partial ordered set of actions

Code Generation for Synchronous Block-diagram _________________ 3/20
Sequential Code Generation

Build a static schedule from a partial ordered set of actions

(proc Step () {
    a;
    b;
    get;
    f;
    set;
    j;
    x;
    h;
    y;
})

(partially) ordered set of actions

(one of the) correct sequential code
Modularity and Feedback

Modularity: a user defined node can be reused in another network

The problem with feedback loops

- this feedback is correct in a *parallel implementation*
- no *sequential single step procedure* can be used
Modularity and Feedback: classical approaches

- **Black-boxing**: user-defined nodes are considered as *instantaneous*, whatever be their actual input/output dependencies
  - compilation is modular
  - rejects causally correct feed-back;
  - E.g., Lucid Synchrone, SCADE, Simulink

- **White-boxing**: nodes are recursively *inline*ed in order to schedule only atomic nodes
  - Any correct feed-back is allowed but modular compilation is lost
  - E.g., Academic Lustre compiler; on user demand in SCADE via *inline* directives.

- **Grey-boxing?**
Grey-boxing

Some actions can be gathered without forbidding correct feedback loops:

- find such a \textit{(minimal) set of blocks} together with their inter-dependencies:
  this is called the \textit{(Optimal) Static Scheduling Problem}

- only need to inline the \textit{blocks dependency graph} within the caller
Grey-boxing

Some actions can be gathered without forbidding correct feedback loops:

- find such a \textit{(minimal) set of blocks} together with their inter-dependencies: this is called the \textit{(Optimal) Static Scheduling Problem}

- only need to inline the \textit{blocks dependency graph} within the caller
Grey-boxing

Some actions can be gathered without forbidding correct feedback loops:

- find such a (minimal) *set of blocks* together with their inter-dependencies:
  this is called the *(Optimal) Static Scheduling Problem*

- only need to inline the *blocks dependency graph* within the caller

dependency analysis
Grey-boxing

Some actions can be gathered without forbidding correct feedback loops:

- find such a (minimal) set of blocks together with their inter-dependencies:
  this is called the (Optimal) Static Scheduling Problem

- only need to inline the blocks dependency graph within the caller

![Diagram of block dependencies]
Grey-boxing

Some actions can be gathered without forbidding correct feedback loops:

- find such a *(minimal)* set of blocks together with their inter-dependencies:
  this is called the *(Optimal) Static Scheduling Problem*

- only need to inline the *blocks dependency graph* within the caller

```
proc P1 () {
    a; get;
    b; f;
    h; y;
}
proc P2 () {
    a; set;
    j; x;
}
P1 before P2
```

Code Generation for Synchronous Block-diagram
State of the Art

- Separate compilation of LUSTRE [Raymond, 1988]: *non optimal*

- Compilation/code distribution of SIGNAL [Benveniste et al, 90’s]: *more general: conditional scheduling, not optimal*

- More recently, [Lublinerman, Szegedy and Tripakis, POPL’09]: *optimal, proof of NP-hardness, iterative search of the optimal solution through 3-SAT encoding.*
State of the Art

- Separate compilation of LUSTRE [Raymond, 1988]: *non optimal*

- Compilation/code distribution of SIGNAL [Benveniste et al, 90’s]: *more general: conditional scheduling, not optimal*

- More recently, [Lublinerman, Szegedy and Tripakis, POPL'09]:
  *optimal, proof of NP-hardness, iterative search of the optimal solution through 3-SAT encoding."

This work addresses the Optimal Static Scheduling Problem (OSS):

- proposes an encoding of the problem based on input/output analysis which gives:
  - in (most) cases, an optimal solution in polynomial time
  - or a 3-sat simplified encoding.

- practical experiments show that the 3-sat solving is almost never necessary
Formalization of the Problem

**Definition: Abstract Data-flow Networks**

A system \((A, I, O, \preceq)\):

1. a finite set of actions \(A\),
2. a subset of inputs \(I \subseteq A\),
3. a subset of output \(O \subseteq A\) (not necessarily disjoint from \(I\))
4. and a partial order \(\preceq\) to represent precedence relation between actions.

**Definition: Compatibility**

Two actions \(x, y \in A\) are said to be (static scheduling) compatible and this is written \(x \chi y\) when the following holds:

\[
x \chi y \overset{\text{def}}{=} \forall i \in I, \forall o \in O, ((i \preceq x \land y \preceq o) \Rightarrow (i \preceq o)) \land ((i \preceq y \land x \preceq o) \Rightarrow (i \preceq o))
\]

If two nodes are incompatible, gathering them into the same block creates an extra input/output dependency, and then forbids a possible feedback loop.
Formalization of the goal

The goal is to find an *equivalence relation* (the set of blocks) implying compatibility plus a *dependence order* between blocks, that is, a *preorder relation*
Formalization of the goal

The goal is to find an *equivalence relation* (the set of blocks) implying compatibility plus a *dependence order* between blocks, that is, a *preorder relation*

---

**Definition: (Optimal) Static Scheduling**

A static scheduling over \((A, \preceq, I, O)\) is a relation \(\preceq\) satisfying:

\((SS-0)\) \(\preceq\) is a pre-order (reflexive, transitive)

\((SS-1)\) \(x \preceq y \Rightarrow x \preceq y\)

\((SS-2)\) \(\forall i \in I, \forall o \in O, \ i \preceq o \ \Leftrightarrow \ i \preceq o\)
Formalization of the goal

The goal is to find an \textit{equivalence relation} (the set of blocks) implying compatibility plus a \textit{dependence order} between blocks, that is, a \textit{preorder relation}.

\textbf{Definition: (Optimal) Static Scheduling}

A static scheduling over \((A, \preceq, I, O)\) is a relation \(\precsim\) satisfying:

(SS-0) \(\precsim\) is a pre-order (reflexive, transitive)

(SS-1) \(x \preceq y \Rightarrow x \precsim y\)

(SS-2) \(\forall i \in I, \forall o \in O, i \precsim o \iff i \preceq o\)

Corrolary: let \(\precsim\) be a S.S. and \((x \simeq y) \iff (x \precsim y \land y \precsim x)\) the associated equivalence, then \(\simeq\) \textit{implies} \(\chi\).
Formalization of the goal

The goal is to find an equivalence relation (the set of blocks) implying compatibility plus a dependence order between blocks, that is, a preorder relation

**Definition: (Optimal) Static Scheduling**

A static scheduling over \((A, \preceq, I, O)\) is a relation \(\preceq\) satisfying:

**(SS-0)** \(\preceq\) is a pre-order (reflexive, transitive)

**(SS-1)** \(x \preceq y \Rightarrow x \preceq y\)

**(SS-2)** \(\forall i \in I, \forall o \in O, i \preceq o \iff i \preceq o\)

Corrolary: let \(\preceq\) be a S.S. and \((x \simeq y) \iff (x \preceq y \land y \preceq x)\) the associated equivalence, then \(\simeq\) implies \(\chi\).

Moreover, a Static Scheduling is optimal iff:

**(SS-3)** \(\simeq\) has a minimal number of classes.
Theoretical Complexity

- Lublinerman, Szegedy and Tripakis proved OSS to be NP-hard through a reduction to the Minimal Clique Cover (MCC) problem.

- Since the OSS problem is an optimization problem whose associated decision problem is — *does it exist a solution with* $k$ *classes?* —, they solve it iteratively by searching for a solution with $k = 1, 2, \ldots$ such as:

  - for each $k$, encode the decision problem as a Boolean formula;
  - solve it using a SAT solver.

However, real programs do not reveal such complexity.

- This complexity seems to happen for programs with a large number of inputs and outputs with complex and unusual dependences between them.

- Can we identify simple cases by analyzing input/output dependences?
Input/output Analysis

Input (resp. output) pre-orders

Let $I$ (resp. $O$) be the input (resp. output) function:

$$I(x) \subseteq I(y), \text{ noted } x \preceq^I y,$$

which is a valid SS, (inclusion of inputs),

$$O(y) \subseteq O(x), \text{ noted } x \preceq^O y,$$

which is a valid SS. (reverse inclusion of outputs),

It is never the case that $x$ should be computed after $y$ if either:

- $I(x) \subseteq I(y), \text{ noted } x \preceq^I y,$
- $O(y) \subseteq O(x), \text{ noted } x \preceq^O y,$
Input/output preorder

An even more precise preorder can be build by considering input preorder over output preorder:

- \( \mathcal{I}_O(x) = \{ i \in I \mid i \preceq^O x \} \)
- \( x \preceq^I_O y \iff \mathcal{I}_O(x) \subseteq \mathcal{I}_O(y) \),
- \( x \simeq^I_O y \iff \mathcal{I}_O(x) = \mathcal{I}_O(y) \)

N.B. a similar reasoning leads to the output/input preorder.

Properties

- \( \preceq^I_O \) is a valid SS,
- moreover, it is \textit{optimal for the inputs/outputs}:
  \[ \forall x, y \in I \cup O \quad x \simeq^I_O y \iff x \preceq x \chi y \]
- it follows that, in any optimal solution, input/output that are compatible are necessarily in the same class (see proof in the paper)
In any solution, the class of a node can be characterized by a subset of inputs or key: intuitively this key is the set of inputs that are computed before or with the node.

As shown before, the only possible key for an input or output node $x$ is $I_O(x)$

How to formalize what can be the key of an internal node?
In any solution, the class of a node can be characterized by a subset of inputs or key: intuitively this key is the set of inputs that are computed before or with the node.

As shown before, the only possible key for an input or output node $x$ is $\mathcal{I}_O(x)$.

How to formalize what can be the key of an internal node?

**Definition: KI-encoding**

A KI-enc. is function $\mathcal{K} : A \mapsto 2^I$ which associate a key to every node such that:

1. **(KI-1)** $\forall x \in I \cup O; \mathcal{K}(x) = \mathcal{I}_O(x)$
2. **(KI-2)** $\forall x, y \quad x \preceq y \implies \mathcal{K}(x) \subseteq \mathcal{K}(y)$

Moreover:

**(KI-opt)** it is optimal if the image set is minimal.
Solving the KI-encoding

A system of (in)equations with a variable $K_x$ for each $x \in A$:

- $K_x = I(x)$ for $x \in I \cup O$

- $\bigcup_{y \rightarrow x} K_y \subseteq K_x \subseteq \bigcap_{x \rightarrow z} K_z$ otherwise

where $\rightarrow$ is the dependency graph relation (a concise representation of $\preceq$)
KI-encoding vs Static Scheduling

• a solution of KI "is" a solution of SS (modulo key inclusion)

• any solution of SS is not a solution of KI (e.g, $\leq$ itself, in general)

• but, any optimal solution of SS is also an optimal solution of KI (to the absurd, via Input/output preorder).

In other terms: the KI formulation is better than the SS one: it has less solutions, but does not miss any optimal one.
KI-encoding vs Static Scheduling

- a solution of KI "is" a solution of SS (modulo key inclusion)
- any solution of SS is not a solution of KI (e.g, \( \leq \) itself, in general)
- but, any optimal solution of SS is also an optimal solution of KI (to the absurd, via Input/output preorder).

In other terms: the KI formulation is better than the SS one: it has less solutions, but does not miss any optimal one.

Complexity of the encoding

- \( O(n \cdot m^2 \cdot (\log m^2)) \) where \( n \) is the number of actions, \( m \) the maximum number of input/outputs.
- That is, \( O(n \cdot m \cdot B(m) \cdot A(m)) \), where \( B \) is the cost of union/intersection between sets and \( A \), the cost of insertion in a set.
Solving the KI-encoding: Example

\[
K_a = \{a, b\} \quad K_b = \{b\} \quad K_x = \{a, b\} \quad K_y = \{b\}
\]

\[
\emptyset \subseteq K_{get} \subseteq K_{set} \cap K_f
\]

\[
K_a \cup K_{get} \subseteq K_{set} \subseteq \{a, b\}
\]

\[
K_b \cup K_{get} \subseteq K_f \subseteq K_j
\]

\[
K_a \cup K_f \subseteq K_j \subseteq K_x
\]

\[
K_b \subseteq K_h \subseteq K_y
\]

- The system to solve:

  - a variable \( K_x \) for each key

  - input/output keys are mandatory

  - set intervals for others
Solving the KI-encoding: Example

\[ K_a = \{a, b\} \quad K_b = \{b\} \quad K_x = \{a, b\} \quad K_y = \{b\} \]

\[ \emptyset \subseteq K_{get} \subseteq \{a, b\} \cap K_{set} \cap K_f \]

\[ K_a \cup K_{get} \cup \{a, b\} \subseteq K_{set} \subseteq \{a, b\} \]

\[ K_b \cup K_{get} \cup \{b\} \subseteq K_f \subseteq \{a, b\} \cap K_j \]

\[ K_a \cup K_f \cup \{a, b\} \subseteq K_j \subseteq \{a, b\} \cap K_x \]

\[ K_b \cup \{b\} \subseteq K_h \subseteq \{b\} \cap K_y \]

- Compute lower and upper bounds:

\[ k_x^\bot = \bigcup_{y \rightarrow x} k_y^\bot \quad \text{and} \quad k_x^\top = \bigcap_{x \rightarrow z} k_z^\top \]
Solving the KI-encoding: Example

\[ K_a = \{a, b\} \quad K_b = \{b\} \quad K_x = \{a, b\} \quad K_y = \{b\} \]

\[ \emptyset \subseteq K_{\text{get}} \subseteq \{a, b\} \cap K_f \]

\[ \{a, b\} \subseteq K_{\text{set}} \subseteq \{a, b\} \]

\[ \{b\} \subseteq K_f \subseteq \{a, b\} \]

\[ \{a, b\} \subseteq K_j \subseteq \{a, b\} \]

\[ \{b\} \subseteq K_h \subseteq \{b\} \]

- Compute lower and upper bounds:

\[ k_x^\perp = \bigcup_{y \rightarrow x} k_y^\perp \quad \text{and} \quad k_x^\top = \bigcap_{x \rightarrow z} k_z^\top \]

- Propagate, simplify: new equations, constant intervals, others
Solving the KI-encoding: Example

\[ K_a = \{a, b\} \quad K_b = \{b\} \quad K_x = \{a, b\} \quad K_y = \{b\} \]

\[
\emptyset = K_{get} \\
\{a, b\} = K_{set} \\
\{b\} = K_f \\
\{a, b\} = K_j \\
\{b\} = K_h
\]

• Check for "obvious" solutions:

\[ \mapsto \mathcal{K}^\perp : x \rightarrow k_x^\perp \]

\[ \mapsto \text{strategy: compute as soon as possible} \]

\[ \mapsto \text{not "proven" optimal: } \emptyset \text{ not mandatory} \]
Solving the KI-encoding: Example

\[ K_a = \{ a, b \} \quad K_b = \{ b \} \quad K_x = \{ a, b \} \quad K_y = \{ b \} \]
\[ K_{\text{get}} = \{ a, b \} \]
\[ K_{\text{set}} = \{ a, b \} \]
\[ K_f = \{ a, b \} \]
\[ K_j = \{ a, b \} \]
\[ K_h = \{ b \} \]

• Check for "obvious" solutions:

\[ K^\top : x \rightarrow k_x^\top \]

\[ \text{strategy: compute as late as possible} \]

\[ \text{optimal: all keys are mandatory} \]
Dealing with complex systems

Let $S$ be the simplified system, $X$ be the set of actions whose key is still unknown, $\kappa_1, \ldots, \kappa_c$ be the $c$ mandatory keys:

- try to find a solution with $c + 0$ classes:
  - build the formula: $S \land_{x \in X} \bigvee_{j=1}^{j=c}(Kx = \kappa_j)$
  - call a SAT-solver...

- if it fails, try to find a solution with $c + 1$ classes:
  - introduce a new variable $B_1$,
  - build the formula: $S \land_{x \in X} (\bigvee_{j=1}^{j=c}(Kx = \kappa_j) \lor (Kx = B_1))$
  - call a SAT-solver...

- if it fails, try to find a solution with $c + 2$ classes, etc.
Experimentation

The prototype

- extract dependency informations from a LUSTRE (or SCADE) program
- build the simplified KI-encoded system (polynomial)
- check for obvious solutions (linear)
- if no obvious solution, iteratively call a Boolean solver.

We have considered three benchmarks made of the components coming from:

- the whole SCADE V4 standard library
  - reusable programs, modular compilation is relevant

- two large industrial applications
  - not reusable programs, less relevant
  - but bigger programs, more likely to be complex
### Results Overview

<table>
<thead>
<tr>
<th></th>
<th># prgs</th>
<th># nodes</th>
<th># i/o</th>
<th>cpu</th>
<th>triv.</th>
<th>solved</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCADE lib.</td>
<td>223</td>
<td>av. 12</td>
<td>2 to 9</td>
<td>0.14s</td>
<td>65</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>Airbus 1</td>
<td>27</td>
<td>av. 25</td>
<td>2 to 19</td>
<td>0.025s</td>
<td>8</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>Airbus 2</td>
<td>125</td>
<td>av. 65</td>
<td>2 to 26</td>
<td>0.2s</td>
<td>41</td>
<td>83</td>
<td>1*</td>
</tr>
</tbody>
</table>

- as expected: programs in SCADE lib. are (small) and then simple
- but also in Airbus, even with "big" interface
- 1*: not really "complex" (solved by a heuristic: intersection of $k^T_x$)
- the whole test takes 0.35 seconds (CoreDuo 2.8Ghz, MacOS X); 350 LO(Caml).
Conclusion

- Optimal Static Scheduling is theoretically NP-hard

- thus it could be solved, through a suitable encoding, with a general purpose Sat-solver

- A polynomial analysis of inputs/outputs can give:
  - non trivial lower and upper bounds on the number of classes
  - a proved optimal solution in some cases
  - a optimized SAT-encoding that emphazises the sources of complexity

- Experiments show that complex instances are hard to find in real examples

Reference:

Marc Pouzet and Pascal Raymond, Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation. In ACM Int. Conf. on Embedded Software (EMSOFT), oct. 2009.