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 (\(\text{set}\)) and update (\(\text{get}\))

  → reverse dependency (and allow feedback)
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

(partially) ordered set of actions
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 inlined 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 (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
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
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
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
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
```

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

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 \textit{equivalence relation} (the set of blocks) implying compatibility plus a \textit{dependence order} between blocks, that is, a \textit{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) \(\forall i \in I, \forall o \in O, \ i \preceq o \iff i \preceq o\)
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 \(\chi\).
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 \(\succeq\) satisfying:

1. \((SS-0)\) \(\succeq\) is a pre-order (reflexive, transitive)
2. \((SS-1)\) \(x \preceq y \implies x \succeq y\)
3. \((SS-2)\) \(\forall i \in I, \forall o \in O, \ i \succeq o \iff i \preceq o\)

**Corrolary:** let \(\succeq\) be a S.S. and \((x \simeq y) \iff (x \succeq y \land y \succeq x)\) 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 \( k = 1, 2, ... \) such as:
  1. For each \( k \), encode the decision problem as a Boolean formula;
  2. 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 \( \mathcal{I} \) (resp. \( \mathcal{O} \)) be the input (resp. output) function:

It is never the case that \( x \) should be computed after \( y \) if either:

- \( \mathcal{I}(x) \subseteq \mathcal{I}(y) \), noted \( x \preceq^I y \), which is a valid of SS, (inclusion of inputs),
- \( \mathcal{O}(y) \subseteq \mathcal{O}(x) \), noted \( x \preceq^O y \), which is a valid SS. (reverse inclusion of outputs),
Input/output preorder

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

\[ \mathcal{I}_O(x) = \{ i \in I \mid i \sim^O x \} \]

\[ x \sim^{IO} y \iff \mathcal{I}_O(x) \subseteq \mathcal{I}_O(y) , \]

\[ x \simeq^{IO} y \iff \mathcal{I}_O(x) = \mathcal{I}_O(y) \]

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

Properties

\[ \sim^{IO} \] is a valid SS,

moreover, it is **optimal for the inputs/outputs**:

\[ \forall x, y \in I \cup O \quad x \sim^{IO} y \iff x \chi y \]

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 $\mathcal{I}_O(x)$.

How to formalize what can be the key of an internal node?
Input-Set Encoding

- 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:

(KI-1) $\forall x \in I \cup O; \mathcal{K}(x) = \mathcal{I}_O(x)$

(KI-2) $\forall x, y \ x \leq y \Rightarrow \mathcal{K}(x) \subseteq \mathcal{K}(y)$

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 $x \in A$:

- $K_x = \mathcal{I}_O(x)$ for $x \in I \cup O$

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

where $\to$ 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., ≤ 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., $\preceq$ 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 $K_1$-encoding: Example

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

$$\emptyset \subseteq K_{\text{get}} \subseteq K_{\text{set}} \cap K_f$$

$$K_a \cup K_{\text{get}} \subseteq K_{\text{set}} \subseteq \{a, b\}$$

$$K_b \cup K_{\text{get}} \subseteq K_f \subseteq K_j$$

$$K_a \cup K_f \subseteq K_j \subseteq K_x$$

$$K_b \subseteq K_h \subseteq K_y$$

- The system to solve:
  - $\rightarrow$ a variable $K_x$ for each key
  - $\rightarrow$ input/output keys are mandatory
  - $\rightarrow$ set intervals for others
Solving the KI-encoding: Example

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

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

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

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

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

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

- Compute lower and upper bounds:

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

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

- Check for "obvious" solutions:
  \[ \mapsto K^\perp : x \rightarrow k_x^\perp \]
  \[ \mapsto \text{strategy: compute as soon as possible} \]
  \[ \mapsto \text{not "proven" optimal: } \emptyset \text{ not mandatory} \]
Solving the KI-encoding: Example

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

- Check for "obvious" solutions:
  \[ \mathcal{K}^\top : x \rightarrow k_x^\top \]

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

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

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

- try to find a solution with $c + 0$ classes:
  - build the formula: $S \land_{x \in X} \lor_{j=1}^{j=c} (K_x = \kappa_j)$
  - 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: $S \land_{x \in X} (\lor_{j=1}^{j=c} (K_x = \kappa_j) \lor (K_x = B_1))$
  - call a SAT-solver...

- if it fails, try to find a solution with $c + 2$ classes, etc.
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. (# blocks)</th>
<th>solved (# blocks)</th>
<th>other (# blocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SCADE lib.</strong></td>
<td>223</td>
<td>av. 12</td>
<td>2 to 9</td>
<td>0.14s</td>
<td>65 (1)</td>
<td>158 (1 or 2)</td>
<td></td>
</tr>
<tr>
<td><strong>Airbus 1</strong></td>
<td>27</td>
<td>av. 25</td>
<td>2 to 19</td>
<td>0.025s</td>
<td>8 (1)</td>
<td>19 (1 to 4)</td>
<td></td>
</tr>
<tr>
<td><strong>Airbus 2</strong></td>
<td>125</td>
<td>av. 65</td>
<td>2 to 26</td>
<td>0.2s</td>
<td>41 (1 to 3)</td>
<td>83 (1 to 4)</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 $k_{x}^{T}$)
- 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.