Static Analysis-based Debugging, Certification, Testing, and Optimization with CiaoPP

Manuel Hermenegildo$^{1,2}$

(with F. Bueno,$^2$ G. Puebla,$^2$ M. Carro,$^2$ P. López-García,$^{4,1}$ J. Morales,$^3$ E. Mera,$^3$ J. Navas,$^5$ R. Haemmerlé,$^1$ M. Méndez,$^5$ A. Casas,$^5$ J. Correas,$^3$ E. Albert,$^3$ P. Arenas$^3$)

$^1$IMDEA Software Institute, Spain
$^2$Technical University of Madrid – UPM, Spain
$^3$Complutense University of Madrid – UCM, Spain
$^4$Spanish Research Council (CSIC), Spain
$^5$CS and EECE Depts., U. of New Mexico, USA

TAPAS 2010 (SAS) – Perpignan, France – Sept. 17, 2010
**Objective**

- Facilitate the development of safe, efficient programs.

**Approach:**

- Next-generation, higher-level, *multiparadigm* prog. languages.

- Improved program development environments.

  - A framework (CiaoPP) which integrates:
    - Debugging.
    - Verification and certification.
    - Testing.
    - Optimization (optimized compilation, parallelization, ...).
Objective

- Facilitate the development of safe, efficient programs.

Approach:

- Next-generation, higher-level, multiparadigm prog. languages.

- Improved program development environments.

- A framework (CiaoPP) which integrates:
  - Debugging.
  - Verification and certification.
  - Testing.
  - Optimization (optimized compilation, parallelization, ...).
## Verification, Diagnosis, and Safe Approximations

- Need to compare actual semantics $\llbracket P \rrbracket$ with intended semantics $\mathcal{I}$:

<table>
<thead>
<tr>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>P is partially correct w.r.t. $\mathcal{I}$ iff $\llbracket P \rrbracket \leq \mathcal{I}$</td>
<td>$\llbracket P \rrbracket_{\alpha^+} \leq \mathcal{I}_{\alpha}$</td>
</tr>
<tr>
<td>P is complete w.r.t. $\mathcal{I}$ iff $\mathcal{I} \leq \llbracket P \rrbracket$</td>
<td>$\mathcal{I}<em>{\alpha} \leq \llbracket P \rrbracket</em>{\alpha^-}$</td>
</tr>
<tr>
<td>P is incorrect w.r.t. $\mathcal{I}$ iff $\llbracket P \rrbracket \not\leq \mathcal{I}$</td>
<td>$\llbracket P \rrbracket_{\alpha^+} \not\leq \mathcal{I}<em>{\alpha}$, or $\mathcal{I}</em>{\alpha} \cap \llbracket P \rrbracket_{\alpha} = \emptyset \land \llbracket P \rrbracket_{\alpha} \not= \emptyset$</td>
</tr>
<tr>
<td>P is incomplete w.r.t. $\mathcal{I}$ iff $\mathcal{I} \not\leq \llbracket P \rrbracket$</td>
<td>$\mathcal{I}<em>{\alpha} \not\leq \llbracket P \rrbracket</em>{\alpha^+}$</td>
</tr>
</tbody>
</table>

Usually, partial descriptions of $\mathcal{I}$ available, typically as assertions.

- **Problem:** difficulty computing $\llbracket P \rrbracket$ w.r.t. interesting observables.

- **Approach:** use a safe approximation of $\llbracket P \rrbracket \rightarrow$ i.e., $\llbracket P \rrbracket_{\alpha^+}$ or $\llbracket P \rrbracket_{\alpha^-}$.

- Specially attractive if compiler computes (most of) $\llbracket P \rrbracket_{\alpha^+}$ anyway.

[BDD+97, HPB99, PBH00c, PBH00a, HPBLG03]

Hermenegildo et al. (IMDEA, UPM)  Debug./Cert./Test./Optim. w/CiaoPP  TAPAS Perpignan 17.9.10
Verification, Diagnosis, and Safe Approximations

- Need to compare actual semantics $\llbracket P \rrbracket$ with intended semantics $I$:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is partially correct w.r.t. $I$ iff</td>
<td>$\llbracket P \rrbracket \leq I$</td>
<td>$\llbracket P \rrbracket_{\alpha^+} \leq I_{\alpha}$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $I$ iff</td>
<td>$I \leq \alpha(\llbracket P \rrbracket)$</td>
<td>$I_{\alpha} \leq \llbracket P \rrbracket_{\alpha^-}$</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $I$ iff</td>
<td>$\alpha(\llbracket P \rrbracket) \not\leq I$</td>
<td>$\llbracket P \rrbracket_{\alpha^=} \not\leq I_{\alpha}$, or</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $I$ iff</td>
<td>$I \not\leq \alpha(\llbracket P \rrbracket)$</td>
<td>$I_{\alpha} \not\leq \llbracket P \rrbracket_{\alpha^+}$</td>
</tr>
</tbody>
</table>

Usually, partial descriptions of $I$ available, typically as assertions.

- **Problem:** difficulty computing $\llbracket P \rrbracket$ w.r.t. interesting observables.
- **Approach:** use a safe approximation of $\llbracket P \rrbracket \rightarrow$ i.e., $\llbracket P \rrbracket_{\alpha^+}$ or $\llbracket P \rrbracket_{\alpha^-}$
- Specially attractive if compiler computes (most of) $\llbracket P \rrbracket_{\alpha^+}$ anyway.

[Hermenegildo et al. (IMDEA, UPM)] Debug./Cert./Test./Optim. w/CiaoPP
Verification, Diagnosis, and Safe Approximations

- Need to compare actual semantics $\llbracket P \rrbracket$ with intended semantics $I$:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is partially correct w.r.t. $I$ if</td>
<td>$\alpha(\llbracket P \rrbracket) \leq I_\alpha$</td>
<td>$\llbracket P \rrbracket_{\alpha^+} \leq I_\alpha$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $I$ if</td>
<td>$I_\alpha \leq \alpha(\llbracket P \rrbracket)$</td>
<td>$I_\alpha \leq \llbracket P \rrbracket_{\alpha^=} $</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $I$ if</td>
<td>$\alpha(\llbracket P \rrbracket) \not\leq I_\alpha$</td>
<td>$\llbracket P \rrbracket_{\alpha^=} \not\leq I_\alpha$, or $\llbracket P \rrbracket_{\alpha^+} \cap I_\alpha = \emptyset \land \llbracket P \rrbracket_{\alpha^+} \neq \emptyset$</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $I$ if</td>
<td>$I_\alpha \not\leq \alpha(\llbracket P \rrbracket)$</td>
<td>$I_\alpha \not\leq \llbracket P \rrbracket_{\alpha^+}$</td>
</tr>
</tbody>
</table>

Usually, partial descriptions of $I$ available, typically as assertions.

- **Problem**: difficulty computing $\llbracket P \rrbracket$ w.r.t. interesting observables.

- **Approach**: use a safe approximation of $\llbracket P \rrbracket$ i.e., $\llbracket P \rrbracket_{\alpha^+}$ or $\llbracket P \rrbracket_{\alpha^-}$

- Specially attractive if compiler computes (most of) $\llbracket P \rrbracket_{\alpha^+}$ anyway.

\[ \text{Hermenegildo et al. (IMDEA, UPM)} \quad \text{Debug./Cert./Test./Optim. w/CiaoPP} \]

[BDD$^+$97, HPB99, PBH00c, PBH00a, HPBLG03]
The CiaoPP Framework

Program $P$

:- check
:- trust
:- test $I_{\alpha}$

Builtins/Libs

PREPROCESSOR

Static Analysis

Analysis Info $[[P]]_{\alpha}$

Comparator (Incl. VCgen)

Assertion Normalizer & Lib Itf.

Unit Test

RT Check

:- check
:- false
:- checked

possible run–time error

verification warning

compile–time error

verified

certificate (ACC) + (optimized) code

Definition

<table>
<thead>
<tr>
<th>Condition</th>
<th>$\alpha([P]) \leq I_{\alpha}$</th>
<th>$\alpha([P]) \not\leq I_{\alpha}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is correct w.r.t. $I_{\alpha}$ if</td>
<td>$\alpha([P]) \leq I_{\alpha}$</td>
<td>$\alpha([P]) \not\leq I_{\alpha}$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $I_{\alpha}$ if</td>
<td>$I_{\alpha} \leq \alpha([P])$</td>
<td>$I_{\alpha} \not\leq \alpha([P])$, or $\alpha([P]) \cap I_{\alpha} = \emptyset$</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $I_{\alpha}$ if</td>
<td>$\alpha([P]) \not\leq I_{\alpha}$</td>
<td>$\alpha([P]) \not\leq I_{\alpha}$</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $I_{\alpha}$ if</td>
<td>$I_{\alpha} \not\leq \alpha([P])$</td>
<td>$I_{\alpha} \not\leq \alpha([P])$</td>
</tr>
</tbody>
</table>

[BDD$^+$97, HPB99, PBH00c, PBH00a, HPBLG03, APH05, MLGH09]
The CiaoPP Framework

- Java source / bytecode.
- Ciao (multi-paradigm):
  - Functions (including higher-order, types, etc.).
  - Predicates (unification, search, including ISO-Prolog).
  - Constraints.
  - Objects and imperative constructs.

[BDD\textsuperscript{+}97, HPB99, PBH00c, PBH00a, HPBLG03, APH05, MLGH09]
The Assertion Language

- Assertions optional, can be added at any time.
- Sets of pre/post/global triples (+ “status” field, documentation, ...).
- Use everywhere, for many purposes (including doc generation).
- Make it worthwhile to the programmer to include them.
- Part of the programming language and “runnable” (Ciao).

[BDD+97, PBH97, HPB99, PBH00b, MLGH09]
The Assertion Language (*simplified* grammar, Java)

\[
\begin{align*}
\langle \text{primitive\_assrt} \rangle & ::= \text{primitive\_name}(\text{var}^*)\langle \text{assrt} \rangle^* \\
\langle \text{assrt} \rangle & ::= @\text{requires} (\langle \text{prop} \rangle^*) \\
& | @\text{ensures} (\langle \text{prop} \rangle^*) \\
& | @\text{cost} (\langle \text{resource\_usage} \rangle^*) \\
& | @\text{if} (\langle \text{prop} \rangle^*) \{ \langle \text{prop} \rangle^* \} \ [ \text{cost} (\langle \text{resource\_usage} \rangle^*) ] \\
\langle \text{resource\_usage} \rangle & ::= \text{res\_usage}(\text{res\_name},\langle \text{expr} \rangle) \\
\langle \text{prop} \rangle & ::= \text{type} \\
& | \text{size}(\text{var},\langle \text{sz\_metric} \rangle,\langle \text{expr} \rangle) \\
& | \text{size\_metric}(\text{var},\langle \text{sz\_metric} \rangle) \\
\langle \text{expr} \rangle & ::= \langle \text{expr} \rangle\langle \text{bin\_op} \rangle\langle \text{expr} \rangle \mid (\sum \mid \prod)\langle \text{expr} \rangle \\
& | \langle \text{expr} \rangle\langle \text{expr} \rangle \mid \log\text{num}\langle \text{expr} \rangle \mid -\langle \text{expr} \rangle \\
& | \langle \text{expr} \rangle! \mid \infty \mid \text{num} \\
& | \text{size}([\langle \text{sz\_metric} \rangle,\text{arg}(\text{r\ num})]) \\
\langle \text{bin\_op} \rangle & ::= + \mid - \mid \times \mid / \mid \% \\
\langle \text{sz\_metric} \rangle & ::= \text{int} \mid \text{ref} \mid \ldots
\end{align*}
\]
The Assertion Language (Grammar, Ciao)

\[\text{program\_assrt} ::= \text{:- \{status\_flag\} \{pred\_assrt\}.} \]
\[\text{\quad | :- head\_cost(\{approx\},Res\_name,Δ^H).} \]
\[\text{\quad | :- literal\_cost(\{approx\},Res\_name,Δ^L).}\]

\[\text{\{status\_flag\} ::= \text{trust | check | true | \epsilon}\}
\[\text{\{pred\_assrt\} ::= \text{pred \{pred\_desc\} \{pre\_cond\} \{post\_cond\} \{comp\_cond\}.}\]
\[\text{\{pred\_desc\} ::= \text{Pred\_name | Pred\_name(\{args\})}\]
\[\text{\{args\} ::= \text{Var | Var, \{args\}}\]
\[\text{\{pre\_cond\} ::= \text{:\=\{state\_props\} | \epsilon}\]
\[\text{\{post\_cond\} ::= \text{=:\{state\_props\} | \epsilon}\]
\[\text{\{comp\_cond\} ::= \text{+ \{comp\_props\} | \epsilon}\]
\[\text{\{state\_prop\} ::= \text{size(Var,\{approx\},\{sz\_metric\},\{arith\_expr\}) | State\_prop}\]
\[\text{\{state\_props\} ::= \text{\{state\_prop\} | \{state\_prop\}, \{state\_props\}\]
\[\text{\{comp\_prop\} ::= \text{size\_metric(Var,\{sz\_metric\}) | \{cost\} | Comp\_prop}\]
\[\text{\{comp\_props\} ::= \text{\{comp\_prop\} | \{comp\_prop\}, \{comp\_props\}\]
\[\text{\{cost\} ::= \text{cost(\{approx\},Res\_name,\{arith\_expr\})}\]
\[\text{\{approx\} ::= \text{ub | lb | oub | olb}\]
\[\text{\{sz\_metric\} ::= \text{value | length | size | void}\]
\[\text{\{arith\_expr\} ::= \text{\text{\text{\text{\text{-}}}} \{arith\_expr\} | \{arith\_expr\} \text{!} | \{quantifier\} \{arith\_expr\}\]
\[\text{\quad | \{arith\_expr\} \{bin\_op\} \{arith\_expr\}\]
\[\text{\quad | \{arith\_expr\}\{arith\_expr\} | log\text{Num} \{arith\_expr\}\]
\[\text{\quad | \text{Num | \{sz\_metric\}(Var)}\]
\[\text{\{bin\_op\} ::= \text{+ | - | * | /}\]
\[\text{\{quantifier\} ::= \text{\Sigma | \Pi}\]
The Ciao Assertion Language

:- pred Pred [Precond] [⇒ Postcond] [+ Comp-formula].

Each typically a “mode” of use; the set covers the valid calls.

:- pred qs(X,Y) : list(int) * var ⇒ sorted(Y) + (det,not_fails).

:- pred qs(X,Y) : var * list(int) ⇒ ground(X) + not_fails.

Properties (from libraries or user defined):

:- regtype color := green | blue | red.

:- regtype list(X) := [] | [X|list].

:- prop sorted := [] | [ _ ] | [X,Y|Z] :- X > Y, sorted([Y|Z]).

Program-point Assertions

- Property calls inlined with code: ...., check( X>0 ), ....

Assertion Status

- Each assertion has prefix check, trust, true, false, etc. – its “status.”
The Ciao Assertion Language

:- pred Pred [:Precond] [=> Postcond] [+ Comp-formula ] .
Each typically a “mode” of use; the set covers the valid calls.

:- pred qs(X,Y) : list(int) * var => sorted(Y) + (det,not_fails).
:- pred qs(X,Y) : var * list(int) => ground(X) + not_fails.

Properties (from libraries or user defined):

:- regtype color := green | blue | red.
:- regtype list(X) := [] | [X|list].
:- prop sorted := [] | [ _ ] | [X,Y|Z] :- X > Y, sorted([Y|Z]).

Program-point Assertions

- Property calls inlined with code: ... , check ( X>0 ) , ...

Assertion Status

- Each assertion has prefix check, trust, true, false, etc. –its “status.”
The Analyses

- Modular, parametric, polyvariant abstract interpretation.
- Accelerated, incremental fixpoint.
- Properties:
  - Shapes, data sizes, sharing/aliasing, CHA, determinacy, exceptions, termination, ...
  - Resources (time, memory, energy, ...), (user-defined) resources.
The CiaoPP Tool
The Analyses

Starting Point: The Intermediate Representation

Transformation

Java Source
javac
Java Bytecode
soot + Ciao transform.
Ciao Source

Analysis

IR – CFG (Horn clauses)

Fixpoint algorithm (AI-based)

Sharing
Shape
CHA

Resource Usage
Sets of Pre/Post pairs Prog. Point Info ...

Sizes and Resource Info.

[MLNH07]

Hermenegildo et al. (IMDEA, UPM) Debug./Cert./Test./Optim. w/CiaoPP TAPAS Perpignan 17.9.10 10 / 68
Intermediate Representation

- Used for all analyses: aliasing, CHA, resources, shape/types, data sizes, etc.
- Used to support several languages / paradigms.
- Based on “blocks” (each block represented as a Horn clause).
- E.g., for Java:
  - Elimination of stack variables.
  - Conversion to three-address statements.
  - SSA transformation (e.g., splitting of input/output param).
  - Explicit representation of this and ret as extra block parameters.
  - Conversion of loops into recursions among blocks.
  - Branching, cases, and dynamic dispatch → blocks w/same signature.
  - Generation of block-based CFG.
  - Conversion to horn clauses for easier manipulation.
Example: sending SMSs

```java
public class CellPhone {
    void sendSms(SmsPacket smsPk, Encoder enc, Stream stm) {
        if (smsPk != null) {
            stm.send(enc.format(smsPk.sms));
            sendSms(smsPk.next, enc, stm);
        }
    }

    class SmsPacket {
        String sms;
        SmsPacket next;
    }

    abstract class Stream {
        @Cost({"cents":"2*size(data)"})
        native void send(String data);
    }

    interface Encoder {
        String format(String data);
    }

    class TrimEncoder implements Encoder {
        @Cost({"cents":"0"})
        @Size("size(ret)<=size(s)")
        public String format(String s) {
            return s.trim();
        }
    }

    class UnicodeEncoder implements Encoder {
        @Cost({"cents":"0"})
        @Size("size(ret)<=6*size(s)")
        public String format(String s) {
            return java.net.URLEncoder.encode(s);
        }
    }
}
```
Example: sending SMSs – IR

- **Internal representation:** basic block → Horn clause.
- **Annotations (since Java 1.5)** are preserved in the bytecode so they can be carried over to our IR.
Fixpoint-based Analyzers

![Diagram showing the process of transforming Java Source to Ciao Source and analyzing IR-CFG to derive Fixpoint algorithm (AI-based) for Resource Usage and Sets of Pre/Post pairs Prog. Point Info with Sharing and Shape.]

[MH92, BGH99, PH96, HPMS00, NMLH07]
[MGH94, BCHP96, PH00, BdIBH+01, PCPH06, PCPH08]
An Efficient, Parametric Fixpoint Algorithm

- Computes $\text{lfp}(S^\alpha_P) = \sigma[P]$, s.t. $\sigma[P]$ safely approximates $\sigma[P]$.
- It maintains and computes as a result (simplified):
  - **An answer table**: $\{\text{block} : \lambda_{\text{in}} \mapsto \lambda_{\text{out}}\}$.
    - Exit states for calls to block satisfying precond $\lambda_{\text{in}}$ meet postcond $\lambda_{\text{out}}$.
  - A dependency arc table: $\{A : \lambda_{\text{in}A} \Rightarrow B : \lambda_{\text{in}B}\}$.
    - Answers for call $A : \lambda_{\text{in}A}$ depend on the answers for $B : \lambda_{\text{in}B}$:
      - (if exit for $B : \lambda_{\text{in}B}$ changes, exit for $A : \lambda_{\text{in}A}$ possibly also changes).
    - $\text{Dep}(B : \lambda_{\text{in}B}) = \sigma$ the set of entries depending on $B : \lambda_{\text{in}B}$.

- Characteristics:
  - **Precision**: context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency**: memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity**: abstract domains are plugins, configurable, ...
  - Handles mutually recursive methods.
  - Handles library calls (essential for Java), externals, ...
  - Modular, incremental.

*Generic framework* for implementing analyses / generating certificates.
An Efficient, Parametric Fixpoint Algorithm

- Computes $\text{lfp}(S^\alpha_P) = \langle P \rangle_\alpha$, s.t. $\langle P \rangle_\alpha$ safely approximates $\langle P \rangle$.
- It maintains and computes as a result (simplified):
  - **An answer table**: $\{\text{block} : \lambda_{\text{in}} \mapsto \lambda_{\text{out}}\}$.
    - Exit states for calls to block satisfying precond $\lambda_{\text{in}}$ meet postcond $\lambda_{\text{out}}$.
  - **A dependency arc table**: $\{A : \lambda_{\text{in}A} \Rightarrow B : \lambda_{\text{in}B}\}$.
    - Answers for call $A : \lambda_{\text{in}A}$ depend on the answers for $B : \lambda_{\text{in}B}$:
      (if exit for $B : \lambda_{\text{in}B}$ changes, exit for $A : \lambda_{\text{in}A}$ possibly also changes).
    - $\text{Dep}(B : \lambda_{\text{in}B}) = \text{the set of entries depending on } B : \lambda_{\text{in}B}$.

- Characteristics:
  - **Precision**: context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency**: memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity**: abstract domains are plugins, configurable, ...
  - Handles mutually recursive methods.
  - Handles library calls (essential for Java), externals, ...
  - Modular, incremental.

Generic framework for implementing analyses / generating certificates.
An Efficient, Parametric Fixpoint Algorithm

- Computes \( \text{lfp}(S_P^\alpha) = \beta[P]_\alpha \), s.t. \( \beta[P]_\alpha \) safely approximates \( \beta[P] \).
- It maintains and computes as a result (simplified):
  - **An answer table**: \( \{ \text{block} : \lambda_{\text{in}} \mapsto \lambda_{\text{out}} \} \).
    - Exit states for calls to \text{block} satisfying precond \( \lambda_{\text{in}} \) meet postcond \( \lambda_{\text{out}} \).
  - **A dependency arc table**: \( \{ A : \lambda_{\text{in}A} \Rightarrow B : \lambda_{\text{in}B} \} \).
    - Answers for call \( A : \lambda_{\text{in}A} \) depend on the answers for \( B : \lambda_{\text{in}B} \):
      - (if exit for \( B : \lambda_{\text{in}B} \) changes, exit for \( A : \lambda_{\text{in}A} \) possibly also changes).
    - \( \text{Dep}(B : \lambda_{\text{in}B}) = \) the set of entries depending on \( B : \lambda_{\text{in}B} \).

- Characteristics:
  - **Precision**: context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency**: memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity**: abstract domains are plugins, configurable, ...
  - Handles mutually recursive methods.
  - Handles library calls (essential for Java), externals, ...
  - Modular, incremental.

*Generic framework* for implementing analyses / generating certificates.
CFG traversal

- Blocks are nodes; edges are invocations.
- Top-down traversal of this CFG, starting from entry point.
- Within each block: sequence of builtins, handled in the domain.
- Inter-block calls/edges: project, extend, etc. (next slide).
- As graph is traversed, triples \((block, \lambda_{in}, \lambda_{out})\) are stored for each block in a memo table.
- Memo table entries have status \(\in \{\text{fixpoint}, \text{approx.}, \text{complete}\}\).
- Iterate until all complete.
Interprocedural analysis / recursion support

- **Project** the caller state over the actual parameters,
- find all the **compatible implementations** (blocks),
- **rename** to their formal parameters,

... abstractly execute each compatible block, ...

- calculate the **least upper bound** of the partial results of each block (if “monovariant on success” flag),
- **rename back** to the actual parameters and, finally
- **extend** (reconcile) return state into calling state.
Speeding up convergence

- Analyze non-recursive blocks first, use as starting $\lambda_{\text{out}}$ in recursions.
- Blocks derived from conditionals treated specially (no project or extend operations required).
- The $(\text{block}, \lambda_{\text{in}}, \lambda_{\text{out}})$ tuples act as a cache that avoids recomputation.
- Use strongly-connected components (on the fly).
Domain examples

Quite a number of domains: shape, aliasing, nullity, CHA, polyhedra, data sizes, depth-k, determinacy, termination, non-failure, ...

Set-sharing (non-aliasing + nullity):

- Uses set of sets of variables to approximate all possible sharing that occur at a given program point (plus nullity):

\[ SH_p = \{ \{v_0, v_1\}, \{v_0, v_1, v_2\}, \{v_3\}\} \]

“\(v_0\) may share with \(v_1\) & \(v_2\), or just \(v_1\); \(v_3\) may point to a non-null loc.”

Analysis ensures that \(v_3\) definitely does not share with \(v_0\), or \(v_1\), or \(v_2\).

- Much work optimizing it: using ZBDDs, negative representations, etc.

[MH89, MH91, DLGH97, VB02, BLGH04, LGBH05, NBH06, MSHK07]
[MLH08, MKSH08, MMLH^+08, MHKS08, MKH09, LGBH10]
Sharing, experimental results, memory usage

**BitSet vs. ZBDD-based implementation.**

![Graph: Memory usage of BitSet vs. ZBDD (v\(_d\)=0.28)](image)

[MLLH08]

Hermenegildo et al. (IMDEA, UPM)

Debug./Cert./Test./Optim. w/CiaoPP

TAPAS Perpignan 17.9.10
Shape+set sharing

- Sharing can be combined with *structural* information. Set sharing can talk about *any* pointer (not just local vars). E.g., this linked list:

  ![Linked List Diagram]

  Can be abstracted as (“depth-k”):
  \[
  \{(v_0 = (\text{data}:p_0, \text{next}:p_1)), \{\{v_0, p_0\}, \{v_0, p_1\}\}\}
  \]

  A statement like \(v_1 = v_0.\text{data}\) will result in a final abstract state:
  \[
  \{(v_0 = (\text{data}:p_0, \text{next}:p_1)), \{\{v_0, v_1, p_0\}, \{v_0, p_1\}\}\}
  \]

- General support in framework for domain combinations.
Example: Shape / Heap Dependency (em3d, regions)
Example: Shape / Heap Dependency (em3d, r-w deps)
Example: Shape / Heap Dependency (bh, r-w deps)
### Example: Shape / Heap Dependency (JOlden, SPECjvm98)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>Classes</th>
<th>Methods</th>
<th>Time</th>
<th>Shape</th>
<th>RW Dep</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisort</td>
<td>560</td>
<td>36</td>
<td>348</td>
<td>0.26s</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>mst</td>
<td>668</td>
<td>52</td>
<td>485</td>
<td>0.12s</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>tsp</td>
<td>910</td>
<td>42</td>
<td>429</td>
<td>0.15s</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>em3d</td>
<td>1103</td>
<td>56</td>
<td>488</td>
<td>0.31s</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>perimeter</td>
<td>1114</td>
<td>44</td>
<td>381</td>
<td>0.91s</td>
<td>P</td>
<td>N</td>
</tr>
<tr>
<td>health</td>
<td>1269</td>
<td>59</td>
<td>534</td>
<td>1.25s</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>voronoi</td>
<td>1324</td>
<td>58</td>
<td>549</td>
<td>1.80s</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>power</td>
<td>1752</td>
<td>57</td>
<td>520</td>
<td>0.36s</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>bh</td>
<td>2304</td>
<td>61</td>
<td>576</td>
<td>1.84s</td>
<td>P</td>
<td>Y</td>
</tr>
<tr>
<td>db</td>
<td>1985</td>
<td>68</td>
<td>562</td>
<td>1.42s</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>logic</td>
<td>3960</td>
<td>72</td>
<td>620</td>
<td>48.26s</td>
<td>P</td>
<td>Y</td>
</tr>
<tr>
<td>raytrace</td>
<td>5809</td>
<td>63</td>
<td>506</td>
<td>37.09s</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Hermenegildo et al. (IMDEA, UPM) Debug./Cert./Test./Optim. w/CiaoPP TAPAS Perpignan 17.9.10
Resource Analyzers

Transformation

Java Source

Java Bytecode

Ciao Source

Java parser

javac

soot + Ciao transform.

IR – CFG (Horn clauses)

Analysis

Sharing

Shape

CHA

Fixpoint algorithm (AI-based)

Sets of Pre/Post pairs Prog. Point Info...

Resource Usage

Sizes and Resource Info.

[DLH90, LGHD94, LGHD96, DLGHL94, DLGHL97, NMLGH07, MLGCH08, NMLH08, NMLH09]
Inference of user-defined resource usage

Build analysis which automatically infers upper bounds on the usage that a program makes of a general notion of user-definable resources.

- Examples:
  - Memory, execution time, execution steps, data sizes.
  - Bits sent or received over a socket, SMSs sent or received, accesses to a database, calls to a procedure, files left open, money spent, ...
  - Energy consumed, ...

- Approach:
  1. Programmer defines via *assertions* resource-related properties for basic procedures (e.g., libraries).
  2. System infers the resource usage bounds for rest of program as functions of input data sizes.

- Property clearly undecidable $\Rightarrow$ approximation required (bounds that are safe and also as accurate as possible).

- Applications: performance debugging and verification, proof carrying code, resource-oriented optimization, ...

[NMLGH07, NMLH09]
Inference of user-defined resource usage

Build analysis which automatically infers upper bounds on the usage that a program makes of a general notion of user-definable resources.

- **Examples:**
  - Memory, execution time, execution steps, data sizes.
  - Bits sent or received over a socket, SMSs sent or received, accesses to a database, calls to a procedure, files left open, money spent, ..
  - Energy consumed, ...

- **Approach:**
  1. Programmer defines via *assertions* resource-related properties for basic procedures (e.g., libraries).
  2. System infers the resource usage bounds for rest of program as functions of input data sizes.

- Property clearly undecidable $\rightarrow$ approximation required (bounds that are safe and also as accurate as possible).

- Applications: performance debugging and verification, proof carrying code, resource-oriented optimization, ...

[NMLGH07, NMLH09]
The CiaoPP Tool
Resource Analyzer

### Inference of user-defined resource usage

Build analysis which automatically infers upper bounds on the usage that a program makes of a general notion of **user-definable** resources.

- **Examples:**
  - Memory, execution time, execution steps, data sizes.
  - Bits sent or received over a socket, SMSs sent or received, accesses to a database, calls to a procedure, files left open, money spent, ..
  - Energy consumed, ..

- **Approach:**
  1. Programmer defines via *assertions* resource-related properties for basic procedures (e.g., libraries).
  2. System infers the resource usage bounds for rest of program as **functions of input data sizes**.

- Property clearly undecidable → approximation required (bounds that are safe and also as accurate as possible).
- Applications: performance debugging and verification, proof carrying code, resource-oriented optimization, ..

[NMLGH07, NMLH09]
Inference of user-defined resource usage

Build analysis which automatically infers upper bounds on the usage that a program makes of a general notion of user-definable resources.

Examples:
- Memory, execution time, execution steps, data sizes.
- Bits sent or received over a socket, SMSs sent or received, accesses to a database, calls to a procedure, files left open, money spent, ..
- Energy consumed, ...

Approach:
1. Programmer defines via assertions resource-related properties for basic procedures (e.g., libraries).
2. System infers the resource usage bounds for rest of program as functions of input data sizes.

Property clearly undecidable $\rightarrow$ approximation required (bounds that are safe and also as accurate as possible).

Applications: performance debugging and verification, proof carrying code, resource-oriented optimization, ...

[NMLGH07, NMLH09]
User-definable aspects of the analysis

- A *cost model* defines an *upper/lower bound cost* for primitive operations (e.g., methods, bytecode instructions).
  - Provided by the user, via *the assertion language*.

```java
@Cost("cents","2*size(data)")
public native void Stream.send(java.lang.String data);
```

- Some predefined in system libraries.

For platform-dependent resources such as execution time or energy consumption model needs to consider low level factors.

- Assertions:
  - Also used to provide other inputs to the resource analysis such as argument sizes, size metrics, etc. if needed.
  - Also allow improving the accuracy and scalability of the system.
  - Output of resource analysis also expressed via assertions.
  - Used additionally to state resource-related specifications which allows finding bugs, verifying, certifying, etc.
User-definable aspects of the analysis

- A cost model defines an upper/lower bound cost for primitive operations (e.g., methods, bytecode instructions).
  - Provided by the user, via the assertion language.

```java
@Cost("cents","2*size(data)")
public native void Stream.send(java.lang.String data);
```

- Some predefined in system libraries.
  For platform-dependent resources such as execution time or energy consumption model needs to consider low level factors.

- Assertions:
  - Also used to provide other inputs to the resource analysis such as argument sizes, size metrics, etc. if needed.
  - Also allow improving the accuracy and scalability of the system.
  - Output of resource analysis also expressed via assertions.
  - Used additionally to state resource-related specifications which allows finding bugs, verifying, certifying, etc.
Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures – currently limited to acyclic).
   - *Determinacy* information inferred and used to obtain tighter bounds.
   - *Non-failure* (no exceptions) inferred for non-trivial lower bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine *relative* sizes of variable contents.
     (Size measures are derived from inferred shape information.)

3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument sizes.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.

4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain resource usage.
Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures – currently limited to acyclic).
   - *Determinacy* information inferred and used to obtain tighter bounds.
   - *Non-failure* (no exceptions) inferred for non-trivial lower bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine *relative* sizes of variable contents.
     (Size measures are derived from inferred shape information.)

3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument sizes.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.

4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain resource usage.
Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures –currently limited to acyclic).
   - Determinacy information inferred and used to obtain tighter bounds.
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine relative sizes of variable contents. (Size measures are derived from inferred shape information.)

3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument sizes.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.

4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain resource usage.
Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures – currently limited to acyclic).
   - *Determinacy* information inferred and used to obtain tighter bounds.
   - *Non-failure* (no exceptions) inferred for non-trivial lower bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine relative sizes of variable contents. (Size measures are derived from inferred shape information.)

3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument sizes.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.

4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain resource usage.
Example: sending SMSs

```java
public class CellPhone {
    void sendSms(SmsPacket smsPk, Encoder enc, Stream stm) {
        if (smsPk != null) {
            stm.send(enc.format(smsPk.sms));
            sendSms(smsPk.next, enc, stm);
        }
    }

    class SmsPacket {
        String sms;
        SmsPacket next;
    }

    abstract class Stream {
        @Cost("cents","2*size(data)")
        native void send(String data);
    }

    interface Encoder {
        String format(String data);
    }

    class TrimEncoder implements Encoder {
        @Cost("cents","0")
        @Size("size(ret) <= size(s)")
        public String format(String s) {
            return s.trim();
        }
    }

    class UnicodeEncoder implements Encoder {
        @Cost("cents","0")
        @Size("size(ret) <= 6*size(s)")
        public String format(String s) {
            return java.net.URLEncoder.encode(s);
        }
    }
}
```

Hermenegildo et al. (IMDEA, UPM) Debug./Cert./Test./Optim. w/CiaoPP TAPAS Perpignan 17.9.10 30 / 68
Example (I)

1. System takes by default size of input data: \( \text{size}(\text{smsPk}) = n \).
   - Result will be parametric on this.

2. The number of characters *sent* depends on the formatting done by the different encoders:
   - The user indicates that the encoding in TrimEncoder results in a smaller or equal (output) string.
     ```java
class TrimEncoder implements Encoder{
    @Size("size(ret)<=size(s)")
    public String format(String s){
    }
}
```
   - And that the result of UnicodeEncoder can be up to 6 times larger (\( \backslash uxxxx \)) than the one received.
     ```java
class UnicodeEncoder implements Encoder{
    @Size("size(ret)<=6*size(s)")
    public String format(String s){
    }
}
```
After setting up and solving the size equations the system obtains that the upper bound on the number of characters sent is:

\[ \max(6, 1) \times n = 6 \times n = 6 \times \text{size}(\text{smsPk}) \]

The analysis establishes then (cost) recurrences for every method:

\[
\begin{align*}
\text{Cost}_{\text{sendSms}}(r_0, 0, r_2, r_3) &= 0 \\
\text{Cost}_{\text{sendSms}}(r_0, r_1, r_2, r_3) &= \text{cost of sending a char} \times \text{Cost}_{\text{sendSms}}(r_0, r_1 - 1, r_2, r_3)
\end{align*}
\]

where \( r_0, r_1, r_2, \) and \( r_3 \) represent the size of This, SmsPk, enc, and stm, respectively.

Given that we are charged 2 cents per character sent:

\[
\begin{align*}
\text{Cost}_{\text{sendSms}}(r_0, 0, r_2, r_3) &= 0 \\
\text{Cost}_{\text{sendSms}}(r_0, r_1, r_2, r_3) &= 2 \times 6 \times (r_1 - 1) \times \text{Cost}_{\text{sendSms}}(r_0, r_1 - 1, r_2, r_3)
\end{align*}
\]

and the total cost of the \( \text{sendSms} \) method is \( 6 \times r_1^2 - 6 \times r_1 \) cents.
Some results (Java)

<table>
<thead>
<tr>
<th>Program</th>
<th>Resource(s)</th>
<th>t</th>
<th>Resource Usage Func. / Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>BST</td>
<td>Heap usage</td>
<td>367</td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td>CellPhone</td>
<td>SMS monetary cost</td>
<td>386</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Client</td>
<td>Bytes received and bandwidth required</td>
<td>527</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Dhrystone</td>
<td>Energy consumption</td>
<td>759</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Divbytwo</td>
<td>Stack usage</td>
<td>219</td>
<td>$O(\log_2(n))$</td>
</tr>
<tr>
<td>Files</td>
<td>Files left open and Data stored</td>
<td>649</td>
<td>$O(n \times m)$</td>
</tr>
<tr>
<td>Join</td>
<td>DB accesses</td>
<td>460</td>
<td>$O(n \times m)$</td>
</tr>
<tr>
<td>Screen</td>
<td>Screen width</td>
<td>536</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

- Different complexity functions, resources, types of loops/recursion, etc.
Some results (Ciao)

<table>
<thead>
<tr>
<th>Program</th>
<th>Resource</th>
<th>Usage Function</th>
<th>Metrics</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>client</td>
<td>“bits received”</td>
<td>$\lambda x.8 \cdot x$</td>
<td>length</td>
<td>186</td>
</tr>
<tr>
<td>color_map</td>
<td>“unifications”</td>
<td>39066</td>
<td>size</td>
<td>176</td>
</tr>
<tr>
<td>copy_files</td>
<td>“files left open”</td>
<td>$\lambda x.x$</td>
<td>length</td>
<td>180</td>
</tr>
<tr>
<td>eight_queen</td>
<td>“queens movements”</td>
<td>19173961</td>
<td>length</td>
<td>304</td>
</tr>
<tr>
<td>eval_polynom</td>
<td>“FPU usage”</td>
<td>$\lambda x.2.5x$</td>
<td>length</td>
<td>44</td>
</tr>
<tr>
<td>fib</td>
<td>“arith. operations”</td>
<td>$\lambda x.2.17 \cdot 1.61^x + 0.82 \cdot (-0.61)^x - 3$</td>
<td>value</td>
<td>116</td>
</tr>
<tr>
<td>grammar</td>
<td>“phrases”</td>
<td>24</td>
<td>length/size</td>
<td>227</td>
</tr>
<tr>
<td>hanoi</td>
<td>“disk movements”</td>
<td>$\lambda x.2^x - 1$</td>
<td>value</td>
<td>100</td>
</tr>
<tr>
<td>insert_stores</td>
<td>“accesses Stores”</td>
<td>$\lambda n, m.n + k$</td>
<td>length</td>
<td>292</td>
</tr>
<tr>
<td></td>
<td>“insertions Stores”</td>
<td>$\lambda n, m.n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>perm</td>
<td>“WAM instructions”</td>
<td>$\lambda x.(\sum_{i=1}^x 18 \cdot x!) + (\sum_{i=1}^x 14 \cdot \frac{x!}{i!}) + 4 \cdot x!$</td>
<td>length</td>
<td>98</td>
</tr>
<tr>
<td>power_set</td>
<td>“output elements”</td>
<td>$\lambda x.\frac{1}{2} \cdot 2^{x+1}$</td>
<td>length</td>
<td>119</td>
</tr>
<tr>
<td>qsort</td>
<td>“lists parallelized”</td>
<td>$\lambda x.4 \cdot 2^x - 2x - 4$</td>
<td>length</td>
<td>144</td>
</tr>
<tr>
<td>send_files</td>
<td>“bytes read”</td>
<td>$\lambda x, y.x \cdot y$</td>
<td>length/size</td>
<td>179</td>
</tr>
<tr>
<td>subst_exp</td>
<td>“replacements”</td>
<td>$\lambda x, y.2xy + 2y$</td>
<td>size/length</td>
<td>153</td>
</tr>
<tr>
<td>zebra</td>
<td>“resolution steps”</td>
<td>30232844295713061</td>
<td>size</td>
<td>292</td>
</tr>
</tbody>
</table>
Interesting Resource: Execution Time

- Important: e.g., verification of real-time constraints.
- Very hard in current architectures, (e.g., worst-case cache behavior).
  - Certainly feasible in simple processors and with caches turned off.
  - Our approach is complementary to accurate WCET models, which consider cache behavior, pipeline state, etc. (inputs to us).

Approach:

- Obtain timing model of abstract machine instructions through a one-time profiling phase (results provided as assertions).
  - Includes fitting constants in a function if the execution time depends on the argument’s properties.
- Static cost analysis phase which infers a function which returns (bounds on) the execution time of program for given input data sizes.

[MLGCH08]
Interesting Resource: Execution Time

- Important: e.g., verification of real-time constraints.
- Very hard in current architectures, (e.g., worst-case cache behavior).
  - Certainly feasible in simple processors and with caches turned off.
  - Our approach is *complementary* to accurate WCET models, which consider cache behavior, pipeline state, etc. (inputs to us).

- Approach:
  - Obtain timing model of abstract machine instructions through a one-time profiling phase (results provided as assertions).
    - Includes fitting constants in a function if the execution time depends on the argument’s properties.
  - Static cost analysis phase which infers a function which returns (bounds on) the execution time of program for given input data sizes.

[MLGCH08]
First Phase Output

Cost assertions automatically generated in first phase and stored to make the instruction execution costs available to the static analyzer.

Examples

:- true pred unify_variable(A, B): int(A), int(B)
    + (cost(ub, exectime, 667.07),
       cost(lb, exectime, 667.07)).

:- true pred unify_variable(A, B): var(A), gnd(B)
    + (cost(ub, exectime, 233.3),
       cost(lb, exectime, 233.3)).

:- true pred unify_variable(A, B): list(A), list(B)
    + cost(ub, exectime, 271.58+284.34*length(A)).
## Observed and Estimated Execution Time (Intel)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E</td>
<td>110</td>
<td>110</td>
<td>113</td>
<td>-2.4</td>
<td>-2.4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>69</td>
<td>69</td>
<td>71</td>
<td>-2.3</td>
<td>-2.3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>1525</td>
<td>1525</td>
<td>1576</td>
<td>-3.3</td>
<td>-3.3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>1501</td>
<td>1501</td>
<td>1589</td>
<td>-5.7</td>
<td>-5.7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>2569</td>
<td>2569</td>
<td>2638</td>
<td>-2.7</td>
<td>-2.7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>1875</td>
<td>1875</td>
<td>2027</td>
<td>-7.8</td>
<td>-7.8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>E</td>
<td>1868</td>
<td>1868</td>
<td>1931</td>
<td>-3.3</td>
<td>-3.3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L</td>
<td>43</td>
<td>68</td>
<td>81</td>
<td>-67.2</td>
<td>-17.8</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>3414</td>
<td>3569</td>
<td>3640</td>
<td>-6.4</td>
<td>-2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>L</td>
<td>54</td>
<td>79</td>
<td>91</td>
<td>-54.6</td>
<td>-14.8</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>3414</td>
<td>3694</td>
<td>4011</td>
<td>-16.2</td>
<td>-8.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>L</td>
<td>135</td>
<td>142</td>
<td>124</td>
<td>8.6</td>
<td>13.7</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>7922</td>
<td>2937</td>
<td>2858</td>
<td>120.6</td>
<td>2.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>L</td>
<td>216</td>
<td>138</td>
<td>111</td>
<td>72.3</td>
<td>22.5</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>226</td>
<td>216</td>
<td>162</td>
<td>34.0</td>
<td>29.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Energy Consumption

- **Energy Consumption Analysis:**
  - *Energy consumption model* (available for simple processors): describe upper bound consumption of each bytecode inst. in terms of joules:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Inst. Cost in $\mu$J</th>
<th>Mem. Cost in $\mu$J</th>
<th>Total Cost in $\mu$J</th>
</tr>
</thead>
<tbody>
<tr>
<td>iadd</td>
<td>.957860</td>
<td>2.273580</td>
<td>3.23144</td>
</tr>
<tr>
<td>isub</td>
<td>.957360</td>
<td>2.273580</td>
<td>3.230.94</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Resource analysis generates at compile time equations and returns safe, upper- and lower-bound *energy consumption functions*.

[NMLH08]
Debugging and Verification

### Definition

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is prt. correct w.r.t. $I_\alpha$ if</td>
<td>$\alpha([P]) \leq I_\alpha$</td>
<td>$[P]<em>{\alpha+} \leq I</em>\alpha$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $I_\alpha$ if</td>
<td>$I_\alpha \leq \alpha([P])$</td>
<td>$I_\alpha \leq [P]_{\alpha}$</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $I_\alpha$ if</td>
<td>$\alpha([P]) \not\leq I_\alpha$</td>
<td>$[P]<em>{\alpha} = \not\leq I</em>\alpha$, or $[P]<em>{\alpha+} \cap I</em>\alpha = \emptyset \land [P]_{\alpha} \neq \emptyset$</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $I_\alpha$ if</td>
<td>$I_\alpha \not\leq \alpha([P])$</td>
<td>$I_\alpha \not\leq [P]_{\alpha+}$</td>
</tr>
</tbody>
</table>

### References

[BDD+97, HPB99, PBH00c, PBH00a, HPBLG03, HALGP05, PCPH06, PCPH08, MLGH09]

---

Program $P$

$\vdash$ check $\vdash$ trust $\vdash$ test $I_\alpha$

Builtins/Libs

PREPROCESSOR

Static Analysis

Analysis Info $[[P]]_\alpha$

Comparator (Incl. VCgen)

Assertion Normalizer & Lib Itf.

Unit Test

RT Check

$\vdash$ texec

$\vdash$ check

$\vdash$ false

$\vdash$ checked

verified

Possible run-time error

Verification warning

Compile-time error

Certificate (ACC)

(optimized) code

Hermenegildo et al. (IMDEA, UPM)

Debug./Cert./Test./Optim. w/CiaoPP

TAPAS Perpignan 17.9.10
Parts of assertions not verified statically generate run-time checks.

Diagnosis (for both static and dynamic errors).

Comparison not always trivial. E.g., resource debugging/certification:
- Need to compare functions.
- “Segmented” answers.

[BDD\textsuperscript{+}97, HPB99, PBH00c, PBH00a, HPBLG03, HALGP05, PCPH06, PCPH08, MLGH09]
Non-trivial: e.g., Resource Debugging / Certification

- Approximated and intended semantics given as resource usage functions:
  Monotonic arithmetic functions expressing lower or upper bounds on the resource usage of a predicate depending on input data sizes.

Example of intended semantics (partial specifications):

```
:- check comp nrev(A,B) : (ground(A), list(A), var(B) )
  + (resource(lb, steps, length(A)),
    resource(ub, steps, 1 + exp(length(A), 2)) .
```

- Number of steps $\in [\text{length}(A), 1 + \text{length}(A)^2]$.

[HALGP05, LGDB10]
Resource Usage Verification

- SPECIFICATION UPPER/LOWER BOUNDS (SU/SL)
- SPECIFICATION INTERVALS

Input Data Size

SPECIFICATION INTERVALS

SPECIFICATION UPPER/LOWER BOUNDS (SU/SL)
Resource Usage Verification

![Resource Usage Diagram](image-url)
Resource Usage Verification

RESOURCE USAGE
- **SPECIFICATION UPPER/LOWER BOUNDS (SU/SL)**
- **ANALYSIS UPPER/LOWER BOUNDS (SU / SL)**

- **SPECIFICATION INTERVALS**
- **ANALYSIS INTERVALS**

**AL > SU → INCORRECT**
**AU < SU → INCORRECT**
**UNKNOWN**
**CORRECT**

**AL ≥ SL AND AU ≤ SU**
Certification / Abstraction Carrying Code

Program P

:- check
:- trust
:- test

Builtins/Libs

Static Analysis

Analysis Info [[P]]_α

Comparator (Incl. VCgen)

RT Check

Unit Test

:- check
:- false
:- checked

PREPROCESSOR

possible run-time error

verification warning

compile-time error

verified

certificate (ACC)

(optimized) code

[APH05, HALGP05, AAPH06]
Many challenges:

- Generating the certificates automatically.
- Generating minimal certificates.
- Designing *simple, reliable, and efficient checkers* for the certificates.
Many challenges:

- Generating the certificates automatically.
- Generating minimal certificates.
- Designing \textit{simple, reliable, and efficient checkers} for the certificates.
Abstraction-based Certification, Abstraction-Carrying Code

The Abstraction Carrying Code (ACC) scheme [LPAR04]:

- Program
- Analyzer
- Domain(s)
- VCGen
- PRODUCER

- Safety Policy
- Abstraction
- OK
- Check
- VCGen
- OK
- CONSUMER

\([P]_\alpha = \text{Analysis} = \text{lfp}(\text{analysis\_step})\)

Certificate ⊆ \([P]_\alpha\)

Certificate → Safety Policy

Checker = analysis\_step

Many interesting extensions: reduced certificates, incrementality, ...

[APH05, HALGP05, AAPH06]
Abstraction-based Certification, Abstraction-Carrying Code

The Abstraction Carrying Code (ACC) scheme [LPAR04]:

\[ [P]_\alpha = \text{Analysis} = \text{lfp}(analysis\_step) \]

Certificate \( \subset [P]_\alpha \)

Certificate \( \to \) Safety Policy

Checker = analysis\_step

Many interesting extensions: reduced certificates, incrementality, ...

[APH05, HALGP05, AAPH06]
Integration of testing

**Program P**

- :- check
- :- trust
- :- test

**Builtins/Libs**

Static Analysis

- Analysis Info [[P]]

Assertion Normalizer & Lib Itf.

- Comparator (Incl. VCgen)

- Unit Test

- RT Check

PREPROCESSOR

- :- texec
- :- check
- :- false
- :- checked

verified

certificate (ACC)

(optimized) code

possible run-time error

verification warning

compile-time error

[MLGH09]
Supporting unit testing

Assertion schema used:

\[
\text{:- test } \text{Pred} \text{[} :\text{Precond} \text{]} [\Rightarrow \text{Postcond}] [+\text{CompExecProps}].
\]

Such test assertions translate into:

What needs to be checked (normal assertions):

\[
\text{:- check } \text{pred Pred} \text{[} :\text{Precond} \text{]} [\Rightarrow \text{Postcond}] [+\text{CompProps}].
\]

What test case needs to be run (test driver):

\[
\text{:- texec Pred} \text{[} :\text{Precond} \text{]} [+\text{Exec-Formula}].
\]

Many interactions within the integrated framework:

- (Unit) tests are part of the assertion language.
- Parts of unit tests that can be verified at compile-time are deleted.
- Rest of unit testing uses the run-time assertion-checking machinery.
- Unit tests also provide test cases for run-time checks coming from assertions.
- Assertions checked by unit testing, even if not conceived as tests.
Supporting unit testing

Assertion schema used:

```prolog
:- test Pred[:Precond] [=>Postcond] [+CompExecProps].
```

Such test assertions translate into:

What needs to be checked (normal assertions):

```prolog
:- check pred Pred [:Precond] [=>Postcond] [+CompProps].
```

What test case needs to be run (test driver):

```prolog
:- texec Pred [:Precond] [+Exec-Formula].
```

Many interactions within the integrated framework:

- (Unit) tests are part of the assertion language.
- Parts of unit tests that can be verified at compile-time are deleted.
- Rest of unit testing uses the run-time assertion-checking machinery.
- Unit tests also provide test cases for run-time checks coming from assertions.
  - Assertions checked by unit testing, even if not conceived as tests.
Example assertion and testing output:

```prolog
:- test qsort(A,_) : (A=[1,3,2]) => (ground(B),list(num,B),sorted(B)).
```

```
{In /tmp/qsort.pl
ERROR: (lns 32-32) Run-time check failure in assertion for:
 'qsort:qsort'([3,2],[3,2]).
In *success*, unsatisfied property:
 sorted([3,2]).
ERROR: (lns 33-37) Check failed in 'qsort:qsort'([3,2],[3,2])
ERROR: (lns 33-37) Failed when invocation of 'qsort:qsort'([1,3,2],_)
called 'qsort:qsort'([3,2],_) in its body.
}
```

Example application:

- Coded 976 unit tests for ISO compliance of Ciao prolog package.
- Detected large number of previously unknown limitations/errors.
- The tests currently run in under 15 seconds.
Example assertion and testing output:

```prolog
:- test qsort(A,_) : (A=[1,3,2]) => (ground(B),list(num,B),sorted(B)).
```

```
{In /tmp/qsort.pl
ERROR: (lns 32-32) Run-time check failure in assertion for:
  'qsort:qsort'([3,2],[3,2]).
In *success*, unsatisfied property:
  sorted([3,2]).
ERROR: (lns 33-37) Check failed in 'qsort:qsort'([3,2],[3,2])
ERROR: (lns 33-37) Failed when invocation of 'qsort:qsort'([1,3,2],_)
called 'qsort:qsort'([3,2],_) in its body.
}
```

Example application:

- Coded 976 unit tests for ISO compliance of Ciao prolog package.
- Detected large number of previously unknown limitations/errors.
- The tests currently run in under 15 seconds.
Example assertion and testing output:

```prolog
:- test qsort(A,_) : (A=[1,3,2]) => (ground(B),list(num,B),sorted(B)).
```

```prolog
{In /tmp/qsort.pl
ERROR: (lns 32-32) Run-time check failure in assertion for:
  'qsort:qsort'([3,2],[3,2]).
In *success*, unsatisfied property:
  sorted([3,2]).
ERROR: (lns 33-37) Check failed in 'qsort:qsort'([3,2],[3,2])
ERROR: (lns 33-37) Failed when invocation of 'qsort:qsort'([1,3,2],_)
called 'qsort:qsort'([3,2],_) in its body.
}
```

Example application:

- Coded 976 unit tests for ISO compliance of Ciao prolog package.
- Detected large number of previously unknown limitations/errors.
- The tests currently run in under 15 seconds.
Optimizations and Parallelization

Program P

Static Analysis

Analysis Info $[[P]]_\alpha$

Comparation

Comparator

(Incl. VCgen)

PREPROCESSOR

 assertion Normalizer & Lib Itf.

Possible run-time error

Compile-time error

Verification warning

Run-time error

Verified

Certificate (ACC)

(optimized) code

[P] Program

:− check

:− trust

:− test

I_\alpha

Builtins/Libs

Possible run-time error

Compile-time error

Verification warning

Run-time error

Verified

Certificate (ACC)

(optimized) code

[GH91, PH97, PHG99, PAH06] [PH99, MBdlBH99, BGH99, CCH08, MKSH08]

[MCH04, CMM^+06]
Also, optimizations

- Preprocessor architecture useful not just for verification / debugging, but also for optimization:
  - Source-level optimizations:
    - Partial evaluation, (multiple) (abstract) specialization, ...
  - Low-level (WAM) optimizations:
    - Use of specialized instructions.
    - Optimized native code generation.

→ obtaining close-to-C performance for declarative languages (Ciao).

- Parallelization.

- But this is a topic for other talks...

[GH91, PH97, PHG99, PAH06] [MCH04, CMM+06]
Parallelization: Motivation

- Power limits frequency... but available area takes us to tera-device: 1000 billion devices by 2020.
- Move from superscalar to multicore: large (100+) numbers of (possibly less complex) (possibly slower) (possibly heterogeneous) cores.
- Novel performance guarantees and architectural contracts (e.g., the memory model) that may not stabilize for several generations.
- Programmability is a huge challenge.

[PH99, MBdBH99, BGH99, PH97, PH03, CCH08, MKSH08]
(Semi-)Automatic Parallelization and Verification

- Automatic parallelization of traditional languages.
  - Limited because of serial nature but still needs to be done:
    - Large number of existing programs.
    - A paradigm shift will take time.
  - Some success for loops. Now, some more progress in dealing with:
    - Irregular control.
    - Complex, dynamically managed data structures.

Major advances in technology:
- Aliasing analysis, heap shape analysis, heap dependence tracking, ...
- Taking into account (new) memory models (coherence models, transactional, parallel garbage collection, ...).

- Compiler-aided parallelization and verification of parallel programs.
- Automatic parallelization of less serial (more declarative) languages.
  - Significant experience from declarative languages needs to be carried over to these new paradigms (see later).
Parallelization Process

- Conditional dependency graph (of some code segment, e.g., a clause):
  - Vertices: possible tasks (statements, calls,...),
  - Edges: possible dependencies (labels: conds. needed for independence).
- Local or global analysis used to reduce/remove checks in the edges.
- Annotation process converts graph into parallel expressions in source.

```prolog
foo(...) :-
g1(...),
g2(...),
g3(...).
```

```
icond(1−3)
icond(1−2) icond(2−3)
```

```
( test(1−3) −> ( g1, g2 ) & g3
                  ;   g1, ( g2 & g3 ) )
g1, ( g2 & g3 )
```

Alternative:
```
"Annotation"
```

```
Local/Global analysis
and simplification
```

```
g1, ( g2 & g3 )
```

```
Alternative:  g1, ( g2 & g3 )
```

Hermenegildo et al. (IMDEA, UPM)
Debug./Cert./Test./Optim. w/CiaoPP
TAPAS Perpignan 17.9.10
Example:

```prolog
qs([X|L], R) :- part(L, X, L1, L2),
                 qs(L2, R2), qs(L1, R1),
                 app(R1, [X|R2], R).
```

Might be annotated in &-Prolog (or Ciao), using local analysis, as:

```prolog
qs([X|L], R) :-
               part(L, X, L1, L2),
               ( indep(L1, L2) ->
                 qs(L2, R2) & qs(L1, R1)
               ;
                 qs(L2, R2), qs(L1, R1) ),
               app(R1, [X|R2], R).
```

Global analysis would eliminate the `indep(L1, L2)` check.
Some Speedups (for different analysis abstract domains)

![Graph showing speedup vs number of processors for different analysis abstract domains.]

The parallelizer, self-parallelized.

The CiaoPP Tool
Optimizations and Parallelization
Example: Heap Dependency Analysis (em3d, regions)

- Can be used for:
  - Parallelizing programs (independence detection).
  - Verifying/debugging user-provided parallel code.
  - Certifying parallel code.
Example: Heap Dependency Analysis (em3d, r-w deps)
Example: Heap Dependency Analysis (bh, r-w deps)


Granularity Control

- Replace parallel with seq. execution based on task size & overheads.
- Cannot be done completely at compile-time: cost often depends on input (hard to approximate precisely even w/abstract interpretation).

\[ \text{main} :- \text{read}(X), \text{read}(Z), \text{inc\_all}(X,Y) \land r(Z,M), \ldots \]

- Our approach:
  - Derive at compile-time cost functions (to be evaluated at run-time) that efficiently bound task size (lower, upper bounds).
  - Transform programs to carry out run-time granularity control.

- For inc\_all, (assuming “threshold” is 100 units):

\[ \text{main} :- \text{read}(X), \text{read}(Z), ( 2\times \text{length}(X)+1 > 100 \rightarrow \text{inc\_all}(X,Y) \land r(Z,M) ; \text{inc\_all}(X,Y) , r(Z,M) ), \]

[DLH90, LGHD94, LGHD96, DLGHL94, DLGHL97, MLGCH08]
Granularity Control

- Replace parallel with seq. execution based on task size & overheads.
- Cannot be done completely at compile-time: cost often depends on input (hard to approximate precisely even w/abstract interpretation).
  
  \[
  \text{main} :\!\!: \text{read}(X), \text{read}(Z), \text{inc\textunderscore all}(X,Y) \& \text{r}(Z,M), \ldots
  \]

- Our approach:
  - Derive at compile-time cost \textit{functions} (to be evaluated at run-time) that efficiently bound task size (lower, upper \textit{bounds}).
  - Transform programs to carry out run-time granularity control.

  ![Diagram]

  - For \textit{inc\textunderscore all}, (assuming “threshold” is 100 units):
    
    \[
    \text{main} :\!\!: \text{read}(X), \text{read}(Z), \left( 2 \ast \text{length}(X) + 1 > 100 \Rightarrow \text{inc\textunderscore all}(X,Y) \& \text{r}(Z,M) \right) ; \text{inc\textunderscore all}(X,Y) \& \text{r}(Z,M)
    \]

[DLH90, LGHD94, LGHD96, DLGHL94, DLGHL97, MLGCH08]
Refinements: Granularity Control Optimizations

- **Simplification of cost functions:**
  
  $$\ldots, \ (\text{length}(X) > 50 \rightarrow \text{inc\_all}(X,Y) \ & r(Z,M)$$
  
  $$\quad \ ; \ \text{inc\_all}(X,Y)\ , \ r(Z,M)\ ), \ \ldots$$
  
  $$\ldots, \ (\text{length\_gt}(LX,50) \rightarrow \text{inc\_all}(X,Y) \ & r(Z,M)$$
  
  $$\quad \ ; \ \text{inc\_all}(X,Y)\ , \ r(Z,M)\ ), \ \ldots$$

- **Complex thresholds:** use also communication cost functions, load, ...

  **Example:** Assume \(\text{CommCost}(\text{inc\_all}(X)) = 0.1 \ (\text{length}(X) + \text{length}(Y))\)

  We know \(\text{ub\_length}(Y)\) (actually, exact size) = \text{length}(X); thus:

  $$2 \ \text{length}(X) + 1 > 0.1 \ (\text{length}(X) + \text{length}(X)) \cong$$

  $$2 \ \text{length}(X) > 0.2 \ \text{length}(X) \equiv$$

  Guaranteed speedup for any data size!

  $$\iff 2 > 0.2$$

- Checking of data sizes can be stopped once under threshold.
- Data size computations can often be done on-the-fly.
- Static task clustering (loop unrolling), static placement, etc.
Refinements: Granularity Control Optimizations

- **Simplification of cost functions:**
  
  \[
  \ldots, \; (\text{length}(X) > 50 \rightarrow \text{inc\_all}(X,Y) \; \& \; r(Z,M) \\
  \quad ; \; \text{inc\_all}(X,Y) \; , \; r(Z,M) ), \; \ldots
  \]

  \[
  \ldots, \; (\text{length\_gt}(LX,50) \rightarrow \text{inc\_all}(X,Y) \; \& \; r(Z,M) \\
  \quad ; \; \text{inc\_all}(X,Y) \; , \; r(Z,M) ), \; \ldots
  \]

- **Complex thresholds:** use also communication cost functions, load, ...

  **Example:** Assume \(\text{CommCost}(inc\_all(X)) = 0.1 \; (\text{length}(X) + \text{length}(Y))\)

  We know \(ub\_\text{length}(Y)\) (actually, exact size) = \(\text{length}(X)\); thus:

  \[
  2 \; \text{length}(X) + 1 > 0.1 \; (\text{length}(X) + \text{length}(X)) \cong
  
  2 \; \text{length}(X) > 0.2 \; \text{length}(X) \equiv
  
  \text{Guaranteed speedup for any data size!} \quad \Leftarrow \quad 2 > 0.2
  
- **Checking of data sizes** can be stopped once under threshold.
- **Data size computations** can often be done on-the-fly.
- **Static task clustering** (loop unrolling), static placement, etc.
Refinements: Granularity Control Optimizations

- **Simplification of cost functions:**
  
  \[ ..., \text{length}(X) > 50 \rightarrow \text{inc\_all}(X,Y) \land r(Z,M) \]
  
  \[ ; \text{inc\_all}(X,Y), r(Z,M) \], ...
  
  \[ ..., \text{length\_gt}(LX,50) \rightarrow \text{inc\_all}(X,Y) \land r(Z,M) \]
  
  \[ ; \text{inc\_all}(X,Y), r(Z,M) \], ...

- **Complex thresholds:** use also communication cost functions, load, ...

  **Example:** Assume \( \text{CommCost}(\text{inc\_all}(X)) = 0.1 \) \( (\text{length}(X) + \text{length}(Y)) \)

  We know \( \text{ub\_length}(Y) \) (actually, exact size) \( = \text{length}(X) \); thus:

  \[ 2 \text{length}(X) + 1 > 0.1 \text{(length}(X) + \text{length}(X)) \equiv \]

  \[ 2 \text{length}(X) > 0.2 \text{length}(X) \equiv \]

  Guaranteed speedup for any data size! \( \iff 2 > 0.2 \)

- Checking of data sizes can be stopped once under threshold.
- Data size computations can often be done on-the-fly.
- Static task clustering (loop unrolling), static placement, etc.
Refinements: Granularity Control Optimizations

- **Simplification of cost functions:**
  \[
  \ldots, \ ( \text{length}(X) > 50 \rightarrow \text{inc\_all}(X,Y) \& \ r(Z,M) \\
  \quad \ ; \ \text{inc\_all}(X,Y) \ , \ r(Z,M) \ ) , \ldots
  \]
  \[
  \ldots, \ ( \text{length\_gt}(LX,50) \rightarrow \text{inc\_all}(X,Y) \& \ r(Z,M) \\
  \quad \ ; \ \text{inc\_all}(X,Y) \ , \ r(Z,M) \ ) , \ldots
  \]

- **Complex thresholds:** use also communication cost functions, load, ...

**Example:** Assume \( \text{CommCost}(\text{inc\_all}(X)) = 0.1 \ (\text{length}(X) + \text{length}(Y)) \)

We know \( \text{ub\_length}(Y) \) (actually, exact size) = \( \text{length}(X) \); thus:

\[
2 \ \text{length}(X) + 1 > 0.1 \ (\text{length}(X) + \text{length}(X)) \Leftrightarrow \\
2 \ \text{length}(X) > 0.2 \ \text{length}(X) \equiv
\]

Guaranteed speedup for any data size! \( \Leftarrow \ 2 > 0.2 \)

- Checking of data sizes can be stopped once under threshold.
- Data size computations can often be done on-the-fly.
- Static task clustering (loop unrolling), static placement, etc.
Refinements: Granularity Control Optimizations

- Simplification of cost functions:
  
  ..., ( length(X) > 50 -> inc_all(X,Y) & r(Z,M) 
  ; inc_all(X,Y) , r(Z,M) ), ...

  ..., ( length_gt(LX,50) -> inc_all(X,Y) & r(Z,M) 
  ; inc_all(X,Y) , r(Z,M) ), ...

- Complex thresholds: use also communication cost functions, load, ...

  Example: Assume $CommCost(inc\_all(X)) = 0.1 \ (length(X) + length(Y))$

  We know $\text{ub\_length}(Y)$ (actually, exact size) = $length(X)$; thus:

  $$2 \ length(X) + 1 > 0.1 \ (length(X) + length(X)) \cong$$

  $$2 \ length(X) > 0.2 \ length(X) \equiv$$

  Guaranteed speedup for any data size! $\iff 2 > 0.2$

- Checking of data sizes can be stopped once under threshold.
- Data size computations can often be done on-the-fly.
- Static task clustering (loop unrolling), static placement, etc.
Granularity Control System Output Example

g_qsort([], []).
g_qsort([First|L1], L2) :-
    partition3o4o(First, L1, Ls, Lg, Size_Ls, Size_Lg),
    Size_Ls > 20 -> (Size_Lg > 20 -> g_qsort(Ls, Ls2) & g_qsort(Lg, Lg2)
                     ; g_qsort(Ls, Ls2), s_qsort(Lg, Lg2))
    ; (Size_Lg > 20 -> s_qsort(Ls, Ls2), g_qsort(Lg, Lg2)
        ; s_qsort(Ls, Ls2), s_qsort(Lg, Lg2))),
    append(Ls2, [First|Lg2], L2).

partition3o4o(F, [], [], [], 0, 0).
partition3o4o(F, [X|Y], [X|Y1], Y2, SL, SG) :-
    X =< F, partition3o4o(F, Y, Y1, Y2, SL1, SG), SL is SL1 + 1.
partition3o4o(F, [X|Y], Y1, [X|Y2], SL, SG) :-
    X > F, partition3o4o(F, Y, Y1, Y2, SL, SG1), SG is SG1 + 1.
8 processors
8 processors, with granularity control (same scale)
Summary System Timeline

- 1989 - PLAI framework/fixpoint. Set sharing, side-effects.
- 1990 - First cost analysis (for task granularity), upper bounds.
- 1991 - Sharing+Freenes, def (dependencies).
- Early 90’s - Practical auto-parallelization.
- 90’s - Incrementality, modularity, extension to constraint programming, concurrency (dynamic scheduling), domain combinations.
- Mid 90’s, 2006 - Combination with partial evaluation, lower bounds.
- Mid-Late 90’s - CiaoPP: Integrated verification, debugging, optimization.
- Late 90’s - Non-failure (no exceptions), determinacy,
- 2001-... Verification of cost, resources, ...
- 2002-... New shape/type domains, widenings.
- 2003-... Abstraction carrying code, reduced certificates, ...
- 2003-... User-defined resources. Time, energy, ...
- 2005-... Multi-language support, Java, C# (shapes, resources, ...).
Final Remarks

- Instead of full specifications a priori (waterfall):
  - Develop program and specifications gradually, not necessarily in sync.
  - Both can be incomplete (including types!).
    - Temporarily use spec (including tests) as implementation.
  - Go from types, to more complex assertions, to full specifications.
    - Can incrementally strengthen them to defs of full functional correctness.

Safe approximations / abstractions everywhere – essential role!

- View debugging, verification, certification, testing, optimization in an integrated way: strong synergy.
  - E.g., tests as part of the specification.
- FM as integral part of the development cycle (“programmer’s tools”).
  - Need to integrate in standard tool chain.
- Assertion language design is important: many roles, used throughout.
- Assertions, properties in source language; “seamless integration.”
- Multi-language analysis through IR; programs in several languages.
Final Remarks

- Instead of full specifications a priori (waterfall):
  - Develop program and specifications gradually, not necessarily in sync.
  - Both can be incomplete (including types!).
    - Temporarily use spec (including tests) as implementation.
  - Go from types, to more complex assertions, to full specifications.
    - Can incrementally strengthen them to defs of full functional correctness.

  Safe approximations / abstractions everywhere –essential role!

- View debugging, verification, certification, testing, optimization in an integrated way: strong synergy.
  - E.g., tests as part of the specification.

- FMs as integral part of the development cycle (“programmer’s tools”).
  - Need to integrate in standard tool chain.

- Assertion language design is important: many roles, used throughout.
- Assertions, properties in source language; “seamless integration.”
- Multi-language analysis through IR; programs in several languages.
Final Remarks

- Instead of full specifications a priori (waterfall):
  - Develop program and specifications gradually, not necessarily in sync.
  - Both can be incomplete (including types!).
    - Temporarily use spec (including tests) as implementation.
  - Go from types, to more complex assertions, to full specifications.
    - Can incrementally strengthen them to defs of full functional correctness.

Safe approximations / abstractions everywhere –essential role!

- View debugging, verification, certification, testing, optimization *in an integrated way*: strong synergy.
  - E.g., tests as part of the specification.
- FM s as integral part of the development cycle (“programmer’s tools”).
  - Need to integrate in standard tool chain.
- Assertion language design is important: many roles, used throughout.
- Assertions, properties in source language; “seamless integration.”
- Multi-language analysis through IR; programs in several languages.
Final Remarks

- Instead of full specifications a priori (waterfall):
  - Develop program and specifications gradually, not necessarily in sync.
  - Both can be incomplete (including types!).
    - Temporarily use spec (including tests) as implementation.
  - Go from types, to more complex assertions, to full specifications.
    - Can incrementally strengthen them to defs of full functional correctness.
- Safe approximations / abstractions everywhere – essential role!
- View debugging, verification, certification, testing, optimization in an *integrated* way: strong synergy.
  - E.g., tests as part of the specification.
- FM's as integral part of the development cycle (“programmer’s tools”).
  - Need to integrate in standard tool chain.
- Assertion language design is important: many roles, used throughout.
- Assertions, properties in source language; “seamless integration.”
- Multi-language analysis through IR; programs in several languages.
Some Plans for CiaoPP/Ciao:

- Continue multi-language analysis: programs in several languages.
- Increase scalability by exploiting further modularity and incrementality of underlying technology.
- Package parts as reusable components (perhaps part of a more general project?).
- Applications in security (including, e.g., timing attacks).
- Continue with language design (Ciao).
- Continue exploring synergies.
  E.g., combine with theorem proving and/or model checking?
Some Members of The Ciao Forge

- **CiaoPP/Ciao** is really a widely distributed collaborative effort:
  - Plus lots of contributors worldwide: G. Gupta (UT Dallas), E. Pontelli (NM State University), P. Stuckey and M. García de la Banda (Melbourne U.), K. Marriott (Monash U.), M. Bruynooghe, A. Mulkers, G. Janssens, and V. Dumortier (K.U. Leuven), S. Debray (U. of Arizona), J. Maluzynski and W. Drabant, (Linkoping U.), P. Deransart (INRIA), J. Gallagher (Roskilde University), C. Holzbauer (Austrian Research Institute for AI), M. Codish (Beer-Sheva), SICS, ...
References – Overall Model

On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs.

Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program
Validation and Debugging.

Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs.
In Logic-based Program Synthesis and Transformation (LOPSTR’99), number 1817 in LNCS, pages

A Generic Preprocessor for Program Validation and Debugging.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, Analysis and Visualization Tools for

Program Development Using Abstract Interpretation (and The Ciao System Preprocessor).
In 10th International Static Analysis Symposium (SAS’03), number 2694 in LNCS, pages 127–152.
Springer-Verlag, June 2003.

[MLGH09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.
In 25th International Conference on Logic Programming (ICLP’09), number 5649 in LNCS, pages
References – Assertion Language

An Assertion Language for Debugging of Constraint Logic Programs.  

An Assertion Language for Constraint Logic Programs.  

[MLGH09] E. Mera, P. López-García, and M. Hermenegildo.  
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.  

References – Intermediate Representation

A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.  
References – Abstraction Carrying Code


References – Fixpoint-based Analyzers (Abstract Interpreters)

Compile-time Derivation of Variable Dependency Using Abstract Interpretation.

Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming.

Optimized Algorithms for the Incremental Analysis of Logic Programs.

Incremental Analysis of Constraint Logic Programs.

An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.
References – Modular Analysis, Analysis of Concurrency


References – Domains: Sharing/Aliasing


Efficient Set Sharing using ZBDDs.
In 21st Int’l. WS on Languages and Compilers for Parallel Computing (LCPC’08), LNCS.
Springer-Verlag, August 2008.

Identification of Logically Related Heap Regions.
In ISMM’09: Proceedings of the 8th international symposium on Memory management, New York,
NY, USA, June 2009. ACM Press.

References – Domains: Shape/Type Analysis

More Precise yet Efficient Type Inference for Logic Programs.
In International Static Analysis Symposium, volume 2477 of Lecture Notes in Computer Science,

Heap Analysis in the Presence of Collection Libraries.

Efficient context-sensitive shape analysis with graph-based heap models.
In Laurie Hendren, editor, International Conference on Compiler Construction (CC 2008), Lecture
References – Domains: Non-failure, Determinacy

Non-Failure Analysis for Logic Programs.
MIT Press, Cambridge, MA.

Multivariant Non-Failure Analysis via Standard Abstract Interpretation.

Determinacy Analysis for Logic Programs Using Mode and Type Information.

Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.
References – Analysis and Verification of Resources

Task Granularity Analysis in Logic Programs.

Towards Granularity Based Control of Parallelism in Logic Programs.

A Methodology for Granularity Based Control of Parallelism in Logic Programs.

Estimating the Computational Cost of Logic Programs.

Lower Bound Cost Estimation for Logic Programs.

User-Definable Resource Bounds Analysis for Logic Programs.


References – Automatic Parallelization, (Abstract) Partial Evaluation, Other Optimizations

A Technique for Recursive Invariance Detection and Selective Program Specialization.

Abstract Specialization and its Application to Program Parallelization.

Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism.

Abstract Multiple Specialization and its Application to Program Parallelization.

An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework.
Abstract Specialization and its Applications. 
Invited talk.

Abstract Interpretation with Specialized Definitions. 

A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism. 
In M. García de la Banda and E. Pontelli, editors, 24th International Conference on Logic Programming (ICLP’08), volume 5366 of LNCS, pages 651–666. Springer-Verlag, December 2008.

Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models. 
In 21st Int’l. WS on Languages and Compilers for Parallel Computing (LCPC’08), LNCS. Springer-Verlag, August 2008.

Improving the Compilation of Prolog to C Using Modeled Types and Determinism Information. 
In PADL’04, number 3057 in LNCS, pages 86–103. Springer-Verlag, June 2004.

High-Level Languages for Small Devices: A Case Study. 