Bee+Cl@k: An Implementation of Lattice-Based Array Contraction in the Source-to-Source Translator ROSE

Christophe Alias, Fabrice Baray, Alain Darte

COMPSYS Team
http://www.ens-lyon.fr/LIP/COMPSYS
LIP – CNRS – ENS Lyon – UCB Lyon – INRIA, France
Why Contract Arrays?

- Software and hardware memory optimization
Why Contract Arrays?

- Software and hardware memory optimization
Why Contract Arrays?

- Software and hardware memory optimization
Why Contract Arrays?

- Software and hardware memory optimization
Why Contract Arrays?

- **Software** and **hardware** memory optimization
- Reduction of **memory size** and **power consumption**
Why Contract Arrays?

- **Software** and **hardware** memory optimization
- Reduction of **memory size** and **power consumption**
- Optimization for high-level synthesis: reduction of **circuit size**
Why Contract Arrays?

- Software and hardware memory optimization
- Reduction of memory size and power consumption
- Optimization for high-level synthesis: reduction of circuit size
- Complementary optimization for parallelization
Idea

- Reduce the array to the cells living simultaneously

\[
f(0) = f(1) = 1
\]
\[
do \ i = 2, n
\]
\[
\quad f(i) = f(i-1) + f(i-2)
\]
\[
return \ f(n)
\]
Reduce the array to the cells living simultaneously

\[
f(0) = f(1) = 1
\]
\[
do\ i = 2, n
\]
\[
f(i) = f(i-1) + f(i-2)
\]
\[
return\ f(n)
\]
\( f(0) = f(1) = 1 \)

\[
\text{do } i = 2, n \\
\quad f(i) = f(i-1) + f(i-2) \\
\text{return } f(n)
\]

☞ Reduce the array to the cells living simultaneously
\[ f(0) = f(1) = 1 \]
\[
\text{do } i = 2, n \\
\quad f(i) = f(i-1) + f(i-2) \\
\text{return } f(n)
\]

- Reduce the array to the cells living simultaneously

\[
\begin{array}{c}
0 & W & \text{W} \\
1 & W & \text{R1} & \text{R2} \\
2 & f(2) = f(1) + f(0) \\
3 & & & \\
\end{array}
\]
f(0) = f(1) = 1

do i = 2,n
f(i) = f(i-1) + f(i-2)

return f(n)

☞ Reduce the array to the cells living simultaneously
\textbf{Idea}

Reduce the array to the cells living simultaneously

\[
\begin{align*}
    f(0) &= f(1) = 1 \\
    \text{do } i &= 2, n \\
    f(i) &= f(i-1) + f(i-2) \\
    \text{return } f(n)
\end{align*}
\]
**Idea**

- Reduce the array to the cells living simultaneously

\[
f(0) = f(1) = 1
\]
\[
do \ i = 2, n
\]
\[
f(i) = f(i-1) + f(i-2)
\]
\[
\text{return } f(n)
\]
f(0 \mod 2) = f(1 \mod 2) = 1

\textbf{do } i = 2, n
\begin{align*}
  f(i \mod 2) &= f((i-1) \mod 2) + \\
                  &= f((i-2) \mod 2)
\end{align*}
\textbf{return } f(n \mod 2)

\sigma(i) = i \mod 2

\textbf{Idea}

\begin{itemize}
  \item Reduce the array to the cells living simultaneously
\end{itemize}
Outline

- Introduction
- **Problem Statement**
- Contributions
- Array Contraction
- Lifetime Analysis
- Experimental Results
- Conclusion and Future Work
Problem Statement

- **Lifetime of a(i)** = time interval between First Write and Last Read

*Multi-dimensional view of array cell interferences*
Problem Statement

- **Lifetime of** $a(i)$ = time interval between First Write and Last Read
- **Conflict** = when lifetimes overlap
Problem Statement

- **Lifetime of** \( a(i) \) = time interval between First Write and Last Read
- **Conflict** = when lifetimes overlap
- **No conflict** ⇒ can be stored in the same cell!
**Problem Statement**

- **Lifetime of a(i)** = time interval between First Write and Last Read
- **Conflict** = when lifetimes overlap
- **No conflict** ⇒ can be stored in the same cell!
Problem Statement

- **Lifetime of** $a(i)$ = time interval between First Write and Last Read
- **Conflict** = when lifetimes overlap
- **No conflict** ⇒ can be stored in the same cell!
- Find a **correct** mapping $\sigma$ with a **minimal** image set.

\[
\sigma : 1 \rightarrow 1 \\
2 \rightarrow 2 \\
3 \rightarrow 1
\]
**Problem Statement**

- **Lifetime of a(i)** = time interval between First Write and Last Read
- **Conflict** = when lifetimes overlap
- **No conflict** ⇒ can be stored in the same cell!
- Find a **correct** mapping \( \sigma \) with a **minimal** image set.

☞ We look for **linear mappings** \( \sigma(i) = Ai \mod b \) on **regular programs**

\[
\begin{align*}
\sigma : & \quad 1 \rightarrow 1 \\
& \quad 2 \rightarrow 2 \\
& \quad 3 \rightarrow 1
\end{align*}
\]
Outline

• Introduction
• Problem Statement
• Contributions
• Array Contraction
• Lifetime Analysis
• Experimental Results
• Conclusion and Future Work
Contributions

Array Contraction Kernel (Cl@k)

- Contraction method(s) decoupled from program analysis
  - Input = polytope
  - Output = integer lattice (≈ folded bounding box)
Contribution(s) decoupled from program analysis
- Input = polytope
- Output = integer lattice (≈ folded bounding box)

Exact lifetime analysis for arrays
- Instance-wise
- Schedule dependent
Contributions

- **Contraction method(s) decoupled from program analysis**
  - Input = polytope
  - Output = integer lattice ($\approx$ folded bounding box)

- **Exact lifetime analysis for arrays**
  - Instance-wise
  - Schedule dependent

- **Code generation from lattices found by Cl@k**
  - Using the ROSE library, from L. Livermore National Labs
Contributions

- Contraction method(s) decoupled from program analysis
  - Input = polytope
  - Output = integer lattice ($\approx$ folded bounding box)

- Exact lifetime analysis for arrays
  - Instance-wise
  - Schedule dependent

- Code generation from lattices found by Cl@k
  - Using the ROSE library, from L. Livermore National Labs

- Integration in a complete source-to-source translator
  - Experimental results on image processing kernels
Outline

• Introduction
• Position of the Problem
• Contributions
• **Array Contraction**
  - Critical Lattice Method
  - Exhaustive Search
  - Successive Minima
• Lifetime Analysis
• Experimental Results
• Conclusion and Future Work
Critical Lattice Method

- $\sigma$ is correct iff $i$ and $j$ conflicts $\Rightarrow \sigma(i) \neq \sigma(j)$
Critical Lattice Method

- \( \sigma \) is correct iff i and j conflicts \( \Rightarrow \sigma(i) \neq \sigma(j) \)
  
  \[ DS = \{i-j, \text{i and j conflicts}\} \]
Critical Lattice Method

- \( \sigma \) is correct iff \( i \) and \( j \) conflicts \( \Rightarrow \sigma(i) \neq \sigma(j) \)
  \[ DS = \{i-j, \text{i and j conflicts}\} \]
- \( \sigma \) is correct iff \( i-j \in DS, i \neq j \) \( \Rightarrow \sigma(i-j) \neq 0 \)
Critical Lattice Method

- \( \sigma \) is correct iff \( i \) and \( j \) conflicts \( \Rightarrow \) \( \sigma(i) \neq \sigma(j) \)
  \[ DS = \{i-j, i \text{ and } j \text{ conflicts}\} \]

- \( \sigma \) is correct iff \( i-j \in DS, i \neq j \) \( \Rightarrow \) \( \sigma(i-j) \neq 0 \) \( \iff \) \( i-j \notin \ker \sigma \)
Critical Lattice Method

- $\sigma$ is correct iff i and j conflicts $\Rightarrow \sigma(i) \neq \sigma(j)$
  DS = {i-j, i and j conflicts}

- $\sigma$ is correct iff $i-j \in DS$, $i \neq j$ $\Rightarrow \sigma(i-j) \neq 0 \iff i-j \notin \ker \sigma$
  iff $DS \cap \ker \sigma = \{0\} \approx \text{“$\sigma$ injective on DS”}$
**Critical Lattice Method**

- $\sigma$ is correct iff $i$ and $j$ conflicts $\Rightarrow \sigma(i) \neq \sigma(j)$
  
  $\text{DS} = \{i-j, \text{ i and j conflicts}\}$

- $\sigma$ is correct iff $i-j \in \text{DS, i} \neq j$ $\Rightarrow \sigma(i-j) \neq 0 \iff i-j \notin \text{ker } \sigma$

  iff $\text{DS} \cap \text{ker } \sigma = \{0\}$ $\approx \text{“}\sigma \text{ injective on DS”}$

Polytope
(on regular programs)
Critical Lattice Method

- $\sigma$ is correct iff $i$ and $j$ conflicts $\Rightarrow \sigma(i) \neq \sigma(j)$
  
  $DS = \{i-j, i$ and $j$ conflicts\}$

- $\sigma$ is correct iff $i-j \in DS, i \neq j \Rightarrow \sigma(i-j) \neq 0 \iff i-j \notin \ker \sigma$
  
  iff $DS \cap \ker \sigma = \{0\} \approx \text{"$\sigma$ injective on DS"}$

Polytope (on regular programs)
Integer Lattice ($\sigma$ linear)
Critical Lattice Method

• $\sigma$ is correct iff $i$ and $j$ conflicts $\Rightarrow \sigma(i) \neq \sigma(j)$

$DS = \{i-j, i \text{ and } j \text{ conflicts}\}$

$\Rightarrow \sigma$ is correct iff ker $\sigma$ is a strictly admissible lattice of DS
Critical Lattice Method

• $\sigma$ is correct iff $i$ and $j$ conflicts $\Rightarrow \sigma(i) \neq \sigma(j)$
  DS = \{i-j, i and j conflicts\}

  $\Rightarrow \sigma$ is correct iff ker $\sigma$ is a strictly admissible lattice of DS

• contracted size = det ker $\sigma$ (volume of ker $\sigma$)
Critical Lattice Method

• $\sigma$ is correct iff $i$ and $j$ conflicts $\Rightarrow \sigma(i) \neq \sigma(j)$
  $\text{DS} = \{i-j, i \text{ and } j \text{ conflicts}\}$
  $\Rightarrow \sigma$ is correct iff $\ker \sigma$ is a strictly admissible lattice of DS

• contracted size $= \det \ker \sigma$ (volume of $\ker \sigma$)
  $\Rightarrow \sigma$ is optimal iff $\ker \sigma$ is a critical lattice of DS
Critical Lattice Method

- **Problem:** How to find the critical lattice of a polytope?
Critical Lattice Method

- **Problem:** How to find the critical lattice of a polytope?
- **Exhaustive search**
  *Generate-and-test* every possible lattice, starting from a lower bound on det ker $\sigma$
Critical Lattice Method

- **Problem:** How to find the critical lattice of a polytope?

- **Exhaustive search**
  *Generate-and-test* every possible lattice, starting from a lower bound on det ker $\sigma$

- **Heuristics**
  Provide quickly a small (but not optimal) admissible lattice
  - Based on successive minima
Critical Lattice Method

- **Problem:** How to find the critical lattice of a polytope?
- **Exhaustive search**
  - *Generate-and-test* every possible lattice, starting from a *lower bound* on det ker $\sigma$
- **Heuristics**
  - Provide quickly a small (but not optimal) admissible lattice
  - Based on successive minima
  - Based on gauge functions
Outline

- Introduction
- Position of the Problem
- Contributions
- **Array Contraction**
  - Critical Lattice Method
  - **Exhaustive Search**
  - Successive Minima
- Lifetime Analysis
- Experimental Results
- Conclusion and Future Work
Exhaustive Search

- Lower bound [Darte et al., IEEE TC] : \( \det \ker \sigma \geq \frac{\text{Vol}(DS)}{2^n} \)
Exhaustive Search

- **Lower bound** [Darte et al., IEEE TC]: $\det \ker \sigma \geq \frac{\text{Vol}(DS)}{2^n}$

- **Algorithm**: Start from the lower bound, and try every lattice

```plaintext
lower_bound = \frac{\text{Vol}(DS)}{2^n}
for (d = lower_bound; ; d++)
    for each lattice $L$ of volume $d$
        if ($L \cap DS = \{0\}$) return $L$;
```
Exhaustive Search

- **Lower bound** [Darte et al., IEEE TC]: \( \det \ker \sigma \geq \frac{\text{Vol}(DS)}{2^n} \)
- **Algorithm**: Start from the lower bound, and try every lattice

\[
\text{lower_bound} = \frac{\text{Vol}(DS)}{2^n} \\
\text{for} (d = \text{lower_bound}; \; ; \; d++) \\
\quad \text{for each lattice } L \text{ of volume } d \\
\quad \quad \text{if}(L \cap DS = \{0\}) \; \text{return} \; L;
\]
Exhaustive Search

- **Lower bound** [Darte et al., IEEE TC]: \( \det \ker \sigma \geq \frac{\text{Vol}(DS)}{2^n} \)
- **Algorithm**: Start from the lower bound, and try every lattice

```plaintext
lower_bound = Vol(DS) / 2^n
for (d = lower_bound; ; d++)
    for each lattice \( L \) of volume \( d \)
        if \( (L \cap DS = \{0\}) \) return \( L \);
```

---

Linear programming

Hermite normal form
Exhaustive Search

- **Lower bound** [Darte et al., IEEE TC]: \( \text{det } \ker \sigma \geq \frac{\text{Vol}(DS)}{2^n} \)
- **Algorithm**: Start from the lower bound, and try every lattice

```python
lower_bound = \frac{\text{Vol}(DS)}{2^n}
for (d = lower_bound; ; d++)
    for each lattice \( L \) of volume \( d \)
        if \( (L \cap DS = \{0\}) \) return \( L \);
```

- Returns an optimal mapping
Exhaustive Search

- **Lower bound** [Darte et al., IEEE TC]: \( \det \ker \sigma \geq \text{Vol}(DS) / 2^n \)

- **Algorithm**: Start from the lower bound, and try every lattice

  ```
  lower_bound = \text{Vol}(DS) / 2^n 
  for(d = lower_bound; ; d++) 
      for each lattice \( L \) of volume \( d \) 
          if(\( L \cap DS = \{0\} \)) return \( L \);
  ```

- Returns an optimal mapping
- Non-parametrized and expensive \( \Rightarrow \) heuristic(s)
Outline

• Introduction
• Position of the Problem
• Contributions
• Array Contraction
  – Critical Lattice Method
  – Exhaustive Search
  – Successive Minima
• Lifetime Analysis
• Experimental Results
• Conclusion and Future Work
Successive Minima Method

- K 0-symmetric polytope
Successive Minima Method

- K 0-symmetric polytope
- View K as the unit ball of a norm: $||x||_K = \min \{ \lambda > 0, x \in \lambda K \}$
Successive Minima Method

- $K$ 0-symmetric polytope
- View $K$ as the unit ball of a norm: $\|x\|_K = \min \{ \lambda > 0, x \in \lambda K \}$
Successive Minima Method

- K 0-symmetric polytope
- View K as the unit ball of a norm: $||x||_K = \min \{ \lambda > 0, x \in \lambda.K \}$
- Find a basis $X = (x_1 \ldots x_n) \subset \mathbb{Z}^n$ of $\mathbb{R}^n$ with minimal norms
Successive Minima Method

- K 0-symmetric polytope
- View K as the unit ball of a norm: $||x||_K = \min \{ \lambda > 0, x \in \lambda.K \}$
- Find a basis $X = (x_1 \ldots x_n) \subset \mathbb{Z}^n$ of $\mathbb{R}^n$ with minimal norms
  - Here, $x_1 = (0,1)$ and $x_2 = (1,0)$
Successive Minima Method

- $K$ 0-symmetric polytope
- View $K$ as the unit ball of a norm: $\|x\|_K = \min \{ \lambda > 0, x \in \lambda K \}$
- Find a basis $X = (x_1 \ldots x_n) \subset \mathbb{Z}^n$ of $\mathbb{R}^n$ with minimal norms
- $\lambda_i(K) = \|x_i\|_K$ = $i^{th}$ successive minima of $K$
Successive Minima Method

- K 0-symmetric polytope
- View K as the unit ball of a norm: $||x||_K = \min \{ \lambda > 0, x \in \lambda K \}$
- Find a basis $X = (x_1 \ldots x_n) \subset \mathbb{Z}^n$ of $\mathbb{R}^n$ with minimal norms
- $\lambda_i(K) = ||x_i||_K = i^{th}$ successive minima of K
  - Here, $\lambda_1(K) = \lambda_2(K) = 1/2$
Successive Minima Method

- K 0-symmetric polytope
- View K as the unit ball of a norm: $||x||_K = \min \{ \lambda > 0, x \in \lambda K \}$
- Find a basis $X = (x_1 \ldots x_n) \subset \mathbb{Z}^n$ of $\mathbb{R}^n$ with minimal norms
- $\lambda_i(K) = ||x_i||_K = i^{th}$ successive minima of $K$
- Find $\rho_i$ with $\rho_i \lambda_i(K) > 1$ such that $X^+ = (\rho_1 x_1 \ldots \rho_n x_n)$ is a solution
First successive minimum:
For each dimension $i$, find the smallest vector $x$ with $x_i \neq 0$.
then get the global minimum:

$$\lambda_1(K) = \min_{i=1..n} \min \{ \lambda \in \mathbb{Q} \mid \exists x \in \lambda.K \cap \mathbb{Z}^n, x_i > 0 \}$$

$x_1 = \text{corresponding “x”}$
Implementation Issues

- First successive minimum:
  For each dimension $i$, find the smallest vector $x$ with $x_i \neq 0$, then get the global minimum:

$$\lambda_1(K) = \min_{i=1..n} \min \{ \lambda \in \mathbb{Q} | \exists x \in \lambda.K \cap \mathbb{Z}^n, x_i > 0 \}$$

$x_1 = \text{corresponding “x”, mixed ILP}$
Implementation Issues

- **First successive minimum:**
  For each dimension $i$, find the smallest vector $x$ with $x_i \neq 0$, then get the global minimum:

$$\lambda_1(K) = \min_{i=1}^{n} \min \{ \lambda \in \mathbb{Q} \mid \exists x \in \lambda.K \cap \mathbb{Z}^n, x_i > 0 \}$$

  $x_1$ = corresponding “x”, mixed ILP

- **Next successive minima**, assuming $X = (x_1 \ldots x_{i-1})$ found.
Implementation Issues

- **First successive minimum:**
  For each dimension $i$, find the smallest vector $x$ with $x_i \neq 0$, then get the global minimum:

\[
\lambda_1(K) = \min_{i=1..n} \min \{ \lambda \in \mathbb{Q} \mid \exists x \in \lambda.K \cap \mathbb{Z}^n, x_i > 0 \}
\]

$x_1$ = corresponding “x”, mixed ILP

- **Next successive minima**, assuming $X = (x_1 \ldots x_{i-1})$ found.
  Compute $X = UT$ (Hermite normal form)
  $U = (u_1 \ldots u_{i-1} \ u \ldots u_n) \approx$ completion of $X$

  same span than $X$  linearly ind. to $X$
Implementation Issues

- **First successive minimum:**
  For each dimension $i$, find the smallest vector $x$ with $x_i \neq 0$, then get the global minimum:

  $\lambda_1(K) = \min_{i=1..n} \min \{ \lambda \in \mathbb{Q} \mid \exists x \in \lambda.K \cap \mathbb{Z}^n, x_i > 0 \}$

  $x_1 =$ corresponding “$x$”, mixed ILP

- **Next successive minima**, assuming $X = (x_1 \ldots x_{i-1})$ found.
  Compute $X = UT$ (Hermite normal form)
  $U = (u_1 \ldots u_{i-1} u_i \ldots u_n) \approx$ completion of $X$
  Express $K$ in the basis $U$,
  then apply the same process on remaining (lin. ind.) dimensions:

  $\lambda_i(K) = \min_{j=i..n} \min \{ \lambda \in \mathbb{Q} \mid \exists x \in \lambda.UK \cap \mathbb{Z}^n, x_j > 0 \}$

  $x_i =$ corresponding “$x$”
Implementation Issues

- **First successive minimum:**
  For each dimension $i$, find the smallest vector $x$ with $x_i \neq 0$, then get the global minimum:

  \[
  \lambda_1(K) = \min_{i=1..n} \min \{ \lambda \in \mathbb{Q} \mid \exists x \in \lambda.K \cap \mathbb{Z}^n, x_i > 0 \}
  \]

  $x_1$ = corresponding “$x$”, mixed ILP

- **Next successive minima**, assuming $X = (x_1 \ldots x_{i-1})$ found.
  Compute $X = UT$ (Hermite normal form)
  $U = (u_1 \ldots u_{i-1} \ u_i \ldots u_n) \approx$ completion of $X$
  Express $K$ in the basis $U$,
  then apply the same process on remaining (lin. ind.) dimensions:

  \[
  \lambda_i(K) = \min_{j=i..n} \min \{ \lambda \in \mathbb{Q} \mid \exists x \in \lambda.UK \cap \mathbb{Z}^n, x_j > 0 \}
  \]

  $x_i$ = corresponding “$x$”, mixed ILP
Outline

- Introduction
- Position of the Problem
- Contributions
- Array Contraction
- Lifetime Analysis
- Experimental Results
- Conclusion and Future Work
Lifetime Analysis (for Arrays)

- **FW(i)** = first operation writing a(i)
  
  
  
  \[
  \text{do } i = 0, s-1 \\
  S_1 \quad a(i) = 0
  \]

- **LR(i)** = last operation reading a(i)
  
  
  \[
  \text{do } i = s, n \\
  S_2 \quad a(i) = a(i-s) + 1
  \]
Lifetime Analysis (for Arrays)

- \( FW(i) = \text{first operation writing } a(i) \) \\
  \( (S_1,i) \) if \( i = 0..s-1 \) \\
  \( (S_2,i) \) if \( i = s..n \)

- \( LR(i) = \text{last operation reading } a(i) \) \\
  \( (S_2,i+s) \) if \( i = 0..n-s \)

\[
\begin{align*}
\text{do } i &= 0..s-1 \\
S_1 & \quad a(i) = 0
\end{align*}
\]

\[
\begin{align*}
\text{do } i &= s..n \\
S_2 & \quad a(i) = a(i-s) + 1
\end{align*}
\]
Lifetime Analysis (for Arrays)

- FW(i) = first operation writing a(i)
  \((S_1,i)\) if \(i = 0..s-1\)
  \((S_2,i)\) if \(i = s..n\)

- LR(i) = last operation reading a(i)
  \((S_2,i+s)\) if \(i = 0..n-s\)

- Computation of FW and LR is similar to array dataflow analysis

☞ Parametrized ILP (PIP tool)
Lifetime Analysis (for Arrays)

- FW(i) = first operation writing a(i)  
  (S₁,i) if i = 0..s-1  
  (S₂,i) if i = s..n
- LR(i) = last operation reading a(i)  
  (S₂,i+s) if i = 0..n-s
- Computation of FW and LR is similar to array dataflow analysis  
  Parametrized ILP (PIP tool)
- a(i) and a(j) conflicts iff FW(i) < LR(j) and FW(j) < LR(i)
Lifetime Analysis (for Arrays)

- FW(i) = first operation writing a(i)  
  \(S_1, i\) if \(i = 0..s-1\)  
  \(S_2, i\) if \(i = s..n\)

- LR(i) = last operation reading a(i)  
  \(S_2, i+s\) if \(i = 0..n-s\)

- Computation of FW and LR is similar to array dataflow analysis
  - Parametrized ILP (PIP tool)

- a(i) and a(j) conflicts iff FW(i) < LR(j) and FW(j) < LR(i)
  - DS = syntactic combination of clauses from FW and LR
  - Then, project on \(k = i-j\) (Polylib)
Code Generation

- Replace $a(f(i))$ with $a(\sigma(f(i)))$

```plaintext
do i = 0, s-1
    a(i) = 0

do i = s, n
    a(i) = a(i-s) + 1
```
Code Generation

- Replace \( a(f(i)) \) with \( a(\sigma(f(i))) \)

\[
\begin{align*}
do i &= 0, s-1 \\
a(i) &= 0 \\
\sigma(i) &= i \mod s \\
\end{align*}
\[
\begin{align*}
do i &= s, n \\
a(i) &= a(i-s) + 1 \\
a(i \% s) &= 0 \\
\end{align*}
\[
\begin{align*}
do i &= s, n \\
a(i \% s) &= a((i-s) \% s) + 1 \\
\end{align*}
\]
Code Generation

- Replace $a(f(i))$ with $a(\sigma(f(i)))$
- Sequential schedule: STOP

$$\begin{align*}
\text{do } i &= 0, s-1 \\
a(i) &= 0 \\
\text{do } i &= s, n \\
a(i) &= a(i-s) + 1
\end{align*}$$

$$\begin{align*}
\text{do } i &= 0, s-1 \\
\sigma(i) &= i \mod s \\
a(i \% s) &= 0 \\
\text{do } i &= s, n \\
a(i \% s) &= a((i-s) \% s) + 1
\end{align*}$$
Code Generation

- Replace $a(f(i))$ with $a(\sigma(f(i)))$
- Sequential schedule: STOP
- Else generate the code w.r.t. the schedule [Quilleré et al.]

\[
\begin{align*}
\text{do } i &= 0, s-1 \\
& \quad a(i) = 0 \\
\text{do } i &= s, n \\
& \quad a(i) = a(i-s) + 1 \\
\sigma(i) &= i \mod s \\
\text{do } i &= 0, s-1 \\
& \quad a(i\%s) = 0 \\
\text{do } i &= s, n \\
& \quad a(i\%s) = a((i-s)\%s) + 1
\end{align*}
\]
Outline

- Introduction
- Position of the Problem
- Contributions
- Array Contraction
- Lifetime Analysis
- Experimental Results
- Conclusion and Future Work
Experimental Results

- Image processing kernels, sequential schedule
- Pentium III 800 MHz, 256 MB RAM

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Array</th>
<th>Mapping</th>
<th>Storage mapping found</th>
<th>Method</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Compressed</td>
<td>H1</td>
<td>H2</td>
</tr>
<tr>
<td>durbin.c</td>
<td>alpha</td>
<td>100</td>
<td>i → i mod 1</td>
<td>x x x x</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>beta</td>
<td>100</td>
<td>i → i mod 1</td>
<td>x x x x</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>sum</td>
<td>10000</td>
<td>(i, j) → (i mod 1, j mod 1)</td>
<td>x x x x</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>10000</td>
<td>(i, j) → (i mod 100, j mod 2)</td>
<td>x x x x</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(i, j) → (i mod 1, 2i + j mod 197)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reg_detect.c</td>
<td>sum_t</td>
<td>36</td>
<td>(i, j) → (i mod 6, j mod 6)</td>
<td>x x x x</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>mean</td>
<td>36</td>
<td>(i, j) → (i mod 3, i + j mod 9)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>diff</td>
<td>2304</td>
<td>(i, j) → (i mod 6, j mod 6)</td>
<td>x x x x</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(i, j) → (i mod 3, i + j mod 9)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(i, j, k) → (i mod 6, j mod 6, k mod 64)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sum_d</td>
<td>2304</td>
<td>(i, j, k) → (i mod 3, i + j mod 9, k mod 64)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sum_d</td>
<td>2304</td>
<td>(i, j, k) → (i mod 1, j mod 1, k mod 1)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td>dynprog.c</td>
<td>c</td>
<td>100</td>
<td>(i, j) → (i mod 9, j mod 9)</td>
<td>x x x x</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(i, j) → (i mod 1, 13i + j mod 61)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sum_c</td>
<td>10000</td>
<td>(i, j, k) → (i mod 1, j mod 1, k mod 1)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(i, j, k) → (i mod 1, j mod 1, k mod 1)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td>gauss.c</td>
<td>g_acc1</td>
<td>10000</td>
<td>(i, j, k) → (i mod 1, j mod 1, k mod 1)</td>
<td>x x x x</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>g_acc2</td>
<td>10000</td>
<td>(i, j, k) → (i mod 1, j mod 1, k mod 1)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g_tmp</td>
<td>2500</td>
<td>(i, j) → (j mod 50, i mod 48)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(i, j) → (i mod 48, j mod 50)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(i, j) → (j - i mod 2, 24j - 25i mod 1200)</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td>mot_detect_kern.c</td>
<td>Delta</td>
<td>68121</td>
<td>(i, j, k) → (k mod 10, j mod 1, i mod 1)</td>
<td>x x x 3.9</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>ODelta</td>
<td>842</td>
<td>(i, j, k) → (i mod 1, j mod 1, k mod 10)</td>
<td>x x x 10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>i → i mod 1</td>
<td>x x x x</td>
<td>0.4</td>
</tr>
</tbody>
</table>
Outline

- Introduction
- Position of the Problem
- Contributions
- Array Contraction
- Lifetime Analysis
- Experimental Results
- Conclusion and Future Work
Conclusion and Future Work

Conclusion

- Efficient method for array contraction
- Fully implemented in the source-to-source framework ROSE
Conclusion and Future Work

Conclusion

● Efficient method for array contraction
● Fully implemented in the source-to-source framework ROSE

Future Work

● Handle lifetimes depending on parameters
  ☞ Bee: OK, Cl@k: on-going
Conclusion and Future Work

Conclusion

- Efficient method for array contraction
- Fully implemented in the source-to-source framework ROSE

Future Work

- Handle lifetimes depending on parameters
  - Bee: OK, Cl@k: on-going
- Breaking the static control constraint
  - Conservative lifetime analysis
Conclusion and Future Work

Conclusion

- Efficient method for array contraction
- Fully implemented in the source-to-source framework ROSE

Future Work

- Handle lifetimes depending on parameters
  - Bee: OK, Cl@k: on-going
- Breaking the static control constraint
  - Conservative lifetime analysis
- Find a transformation (schedule) optimizing array contraction
  - Extension of unimodular frameworks?
Conclusion

- Efficient method for array contraction
- Fully implemented in the source-to-source framework ROSE

Future Work

- Handle lifetimes depending on parameters
  - Bee: OK, Cl@k: on-going

- Breaking the static control constraint
  - Conservative lifetime analysis

- Find a transformation (schedule) optimizing array contraction
  - Extension of unimodular frameworks?

- Trade-off memory space / execution time
  - Find mappings optimizing data-locality
Conclusion and Future Work

Conclusion

- Efficient method for array contraction
- Fully implemented in the source-to-source framework ROSE

Future Work

- Handle lifetimes depending on parameters
  - Bee: OK, Cl@k: on-going
- Breaking the static control constraint
  - Conservative lifetime analysis
- Find a transformation (schedule) optimizing array contraction
  - Extension of unimodular frameworks?
- Trade-off memory space / execution time
  - Find mappings optimizing data-locality
- Connection with a high-level synthesis tool