Cetus-assisted checkpointing of parallel codes

Gabriel Rodríguez, M.J. Martín, P. González, J. Touriño, R. Doallo

Cetus Users and Compiler Infrastructure Workshop
Galveston, TX, October 2011
Portable checkpointing for SPMD applications.
Aims to provide fully transparent operation.
Preserves application scalability.
Why use a compiler?
Selection of restart-relevant data
Why use a compiler?
 Compile-time coordination

Uncoordinated processes $\rightarrow$ restart inconsistencies
Why use a compiler?
Compile-time coordination
Why use a compiler?
Compile-time coordination
Why Cetus?

- Well, we used SUIF before...
  - Closed-source front-ends.
  - Buggy front-ends.
  - Unmaintained front-ends.
- The Cetus License allows modification and redistribution.
- The Java implementation guarantees portability.
CPPC design

CPPC Compiler (Cetus) → Compiler

Parallel App. (C, C++, Fortran 77, ...) → Fault Tolerant Parallel Application

Adapter (C++)

CPPC Library (C++)
Tested for MPI, although the approach is easily extensible by design.

Similar to a static simulation of the execution.

Uses constant propagation and symbolic expression analysis.

Ignores non-communication statements.
Detect variables relevant to interprocess communications:
- Not to the communicated values, but to the communicating processes.

```c
int MPI_Send( void * buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm )
```
Detect variables relevant to interprocess communications:
- Not to the communicated values, but to the communicating processes.

```c
int dest = (rank + k) % comm_size;
int tag
```

**Semantic input to the compiler**
```c
int MPI_Send( void * buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm )
```
Detect variables relevant to interprocess communications:
- Not to the communicated values, but to the communicating processes.

```c
int MPI_Send( void * buf, int count, MPI_Datatype datatype, 
              int dest, int tag, MPI_Comm comm )
```

```
int dest = (rank + k) % comm_size;
int tag
```

```
int dest, int rank
int tag, int comm_size
int k, ...
```
Detect variables relevant to interprocess communications:
- Not to the communicated values, but to the communicating processes.

Assign known constant values to detected communication-relevant variables.

Analyze the code in execution order.
- Determine whether an instruction is a safe point.
- If it is a communication statement: analyze.
- If it is a communication-relevant statement: symbolic analysis.
- Else, skip to next statement.
Checkpoint insertion

Overview

- Locate points in the code where checkpoints are needed in order to guarantee progress.
- Discard any code not inside loops.
- Computation time cannot be accurately predicted: use heuristics.
Checkpoint insertion
Cost estimation

Procedure \( f() \)

- **(body)**
  - CompoundStatement
  
  \[ w_f = ? \]

- **(call)**
  - ExpressionStatement

  \[ w_{\text{call}} = ? \]

- **(loop)**
  - Loop statement

  \[ w_{\text{loop}} = ? \]

- **(if)**
  - IfStatement

  \[ w_{\text{if}} = ? \]
Checkpoint insertion

Cost estimation

Procedure f()

(body) CompoundStatement \( w_f = ? \)

(call) ExpressionStatement \( w_{call} = w_g \)

FunctionCall \( w_g \)

(loop) Loop statement \( w_{loop} = ? \)

(if) IfStatement \( w_{if} = ? \)
Checkpoint insertion
Cost estimation

\[ w_{call} = w_g \]

\[ w_{loop} = w_1 \]

\[ w_l = \sum_{i=1}^{n} w_{l,i} \]

\[ w_{l,1} \]

\[ w_{l,n} \]
Checkpoint insertion
Cost estimation

Procedure f()

(body)  CompoundStatement

(call)   ExpressionStatement

(loop)  Loop statement

(if)     IfStatement

(then)   CompoundStatement

(else)   CompoundStatement

\( w_f = ? \)

\( w_{\text{call}} = w_g \)

\( w_{\text{loop}} = w_l \)

\( w_{\text{if}} = \frac{w_t + w_e}{2} \)

\( w_t = \sum_{i=1}^{n} w_{t,i} \)

\( w_e = \sum_{i=1}^{n} w_{e,i} \)
Checkpoint insertion
Cost estimation

Procedure f()

(body) CompoundStatement

(call) ExpressionStatement

(loop) Loop statement

(if) IfStatement

\[ w_f = w_{call} + w_{loop} + w_{if} + \ldots \]

\[ w_{call} = w_g \]

\[ w_{loop} = w_l \]

\[ w_{if} = \frac{w_t + w_e}{2} \]
Checkpoint insertion
Loop thresholding

\[ L \]

\[ H \]

\[ l_t \]

\[ d(l_t) \]

\[ h(l) \]
Live variable analysis
Overview

- Analyze sections of code for live variables that need to be stored into checkpoints.
- The traditional analysis proceeds from the end of the code up to the start, traversing basic blocks.
- CPPC does not use the CFG infrastructure in Cetus, but implements an execution order version:
  - Interprocedural version.
  - Some array optimizations.
- Each non compound statement has been annotated with its consumed and generated symbols.
- This information is forward-propagated taking into account the control flow.
Live variable analysis
Traversing the code

\[ \text{LV}_{\text{in}}(BB) = \text{USE}(BB) \cup \{ \text{LV}_{\text{out}}(BB) - \text{DEF}(BB) \} \]

\[ \text{LV}_{\text{out}}(BB) = \bigcup_S \text{LV}_{\text{in}}(S) \]

\[ \text{LV}_{\text{out}}(BB_{\text{end}}) = \emptyset \]
Live variable analysis
Traversing the code

\[
\begin{align*}
\text{consumed} &= \emptyset \\
\text{generated} &= \emptyset \\
\end{align*}
\]

\[
\begin{align*}
\text{consumed} &= \text{consumed} \cup \{USE(s) - \text{generated}\} \\
\text{generated} &= \text{generated} \cup DEF(s) \\
\end{align*}
\]

\[
LV_{\text{in}}(c_i) = \text{consumed}
\]
Putting it all together

"main" FUNCTION

conditional jump

application
code

jump target

var. registers

checkpoint

code analyzed
for live vars.
Putting it all together

"main" FUNCTION
- conditional jump
- application code
- jump target
- "main" registers
- call to f_1
- code analyzed for live vars.

"f_1" FUNCTION
- conditional jump
- application code
- jump target
- "f_1" registers
- call to f_2
- code analyzed for live vars.

"f_n" FUNCTION
- conditional jump
- application code
- jump target
- "f_n" registers
- checkpoint
- code analyzed for live vars.

STACK
- main
- main
- f_1
- ...
Extending Cetus: Fortran support

- Fortran 77 front-end that generates Cetus IR from F77 codes.
- Reuse Cetus IR as much as possible.
- Extend Cetus IR where necessary, preserving interface and behavior.
- Back-end to transform Cetus IR back into F77 code.
Extending Cetus: Fortran support
IR extensions

- **cetus.hir.Declaration**: COMMON, DATA, DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, SAVE.

- **cetus.hir.Literal**: DOUBLE literals.

- **cetus.hir.Specifier**: COMPLEX, DOUBLE COMPLEX, ARRAY(lbound, ubound), CHARACTER*N.

- **cetus.hir.Statement**: Computed GOTOs, FORMAT, Fortran-style DO, Implied DO.

- **cetus.hir.Expression**: expressions in FORMAT, substrings, IO calls.

- **cetus.hir.UnaryOperator**: &&.

- **cetus.hir.BinaryOperator**: ***, //.
# Concluding remarks

## Perceptions on the Cetus infrastructure

<table>
<thead>
<tr>
<th>Perceived strengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Java implementation: portability and clean design.</td>
</tr>
<tr>
<td>- Completely open architecture from head to toe.</td>
</tr>
<tr>
<td>- High level representation.</td>
</tr>
<tr>
<td>- Evolving infrastructure (e.g. new built-in analyses).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Perceived weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Complex IR.</td>
</tr>
<tr>
<td>- Performance.</td>
</tr>
</tbody>
</table>
Questions?

Cetus-assisted checkpointing of parallel codes

Gabriel Rodríguez, M.J. Martín, P. González, J. Touriño, R. Doallo

http://cppc.des.udc.es -- grodriguez@udc.es