Cetus-assisted checkpointing of parallel codes

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

COMPUTER ARCHITECTURE GROUP
UNIVERSITY OF A CORUÑA
GAC.UDC.ES

Cetus Users and Compiler Infrastructure Workshop
Galveston, TX, October 2011

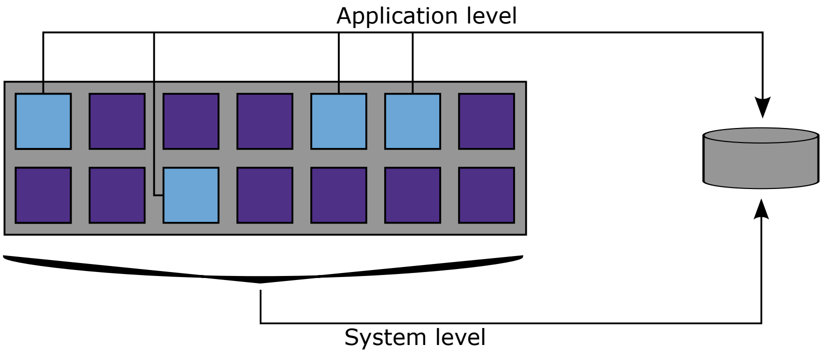
CPPC

ComPiler for Portable Checkpointing

- Portable checkpointing for SPMD applications.
- Aims to provide fully transparent operation.
- Preserves application scalability.

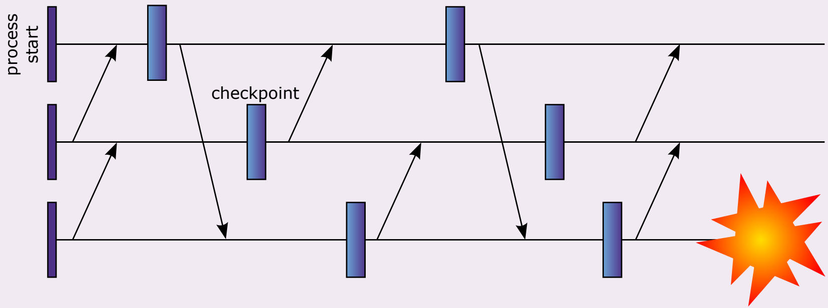
Why use a compiler?

Selection of restart-relevant data



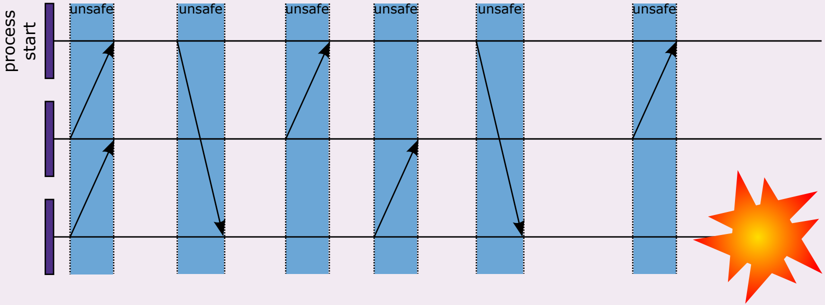
Why use a compiler?
Compile-time coordination

Uncoordinated processes → restart inconsistencies



Why use a compiler?
Compile-time coordination

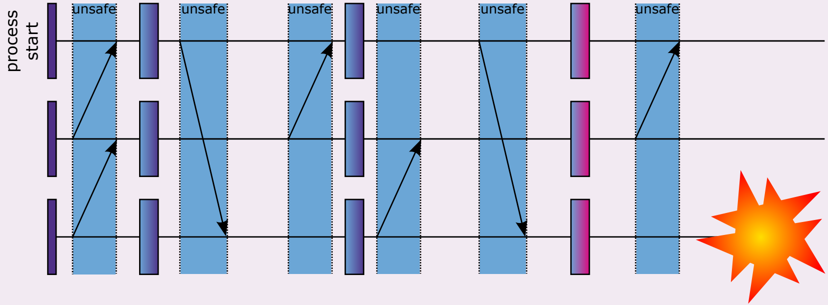
Compile-time coordination



Why use a compiler?

Compile-time coordination

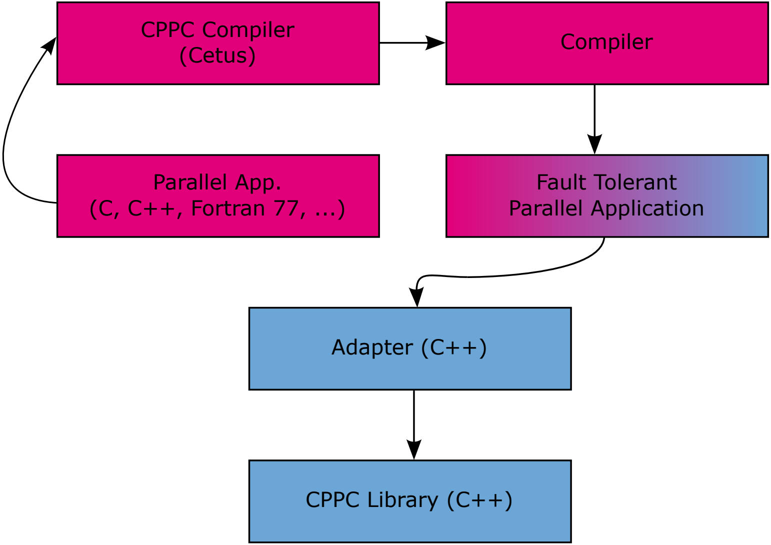
Compile-time coordination



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++)

Communication analysis
Overview

- 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.

Communication analysis
Implementation

1. Detect variables relevant to interprocess communications:
   - Not to the communicated values, but to the communicating processes.

semantic input to the compiler

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

Communication analysis
Implementation

1. Detect variables relevant to interprocess communications:
   - Not to the communicated values, but to the communicating processes.

semantic input to the compiler

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

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

input to the compiler

Communication analysis

Implementation

1. Detect variables relevant to interprocess communications:
   - Not to the communicated values, but to the communicating processes.

semantic input to the compiler

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

```
int dest
int tag
```

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

input to the compiler

```
int dest        int rank
int tag         int comm_size
int k           ...
```

Communication analysis

Implementation

1. Detect variables relevant to interprocess communications:
   - Not to the communicated values, but to the communicating processes.
2. Assign known constant values to detected communication-relevant variables.
3. Analyze the code in execution order.
   1. Determine whether an instruction is a safe point.
   2. If it is a communication statement: analyze.
   3. If it is a communication-relevant statement: symbolic analysis.
   4. Else, skip to next statement.

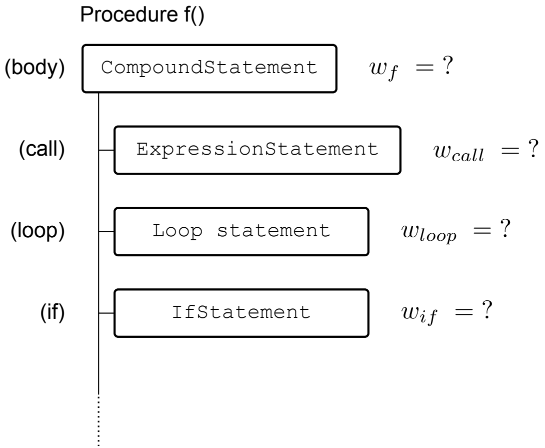
Checkpoint insertion

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.

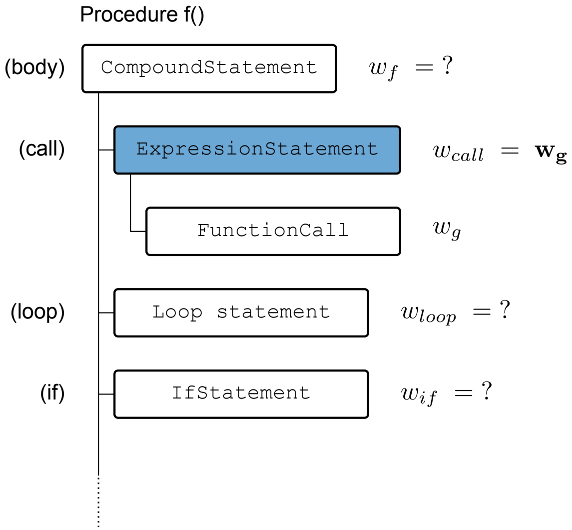
Checkpoint insertion

Cost estimation



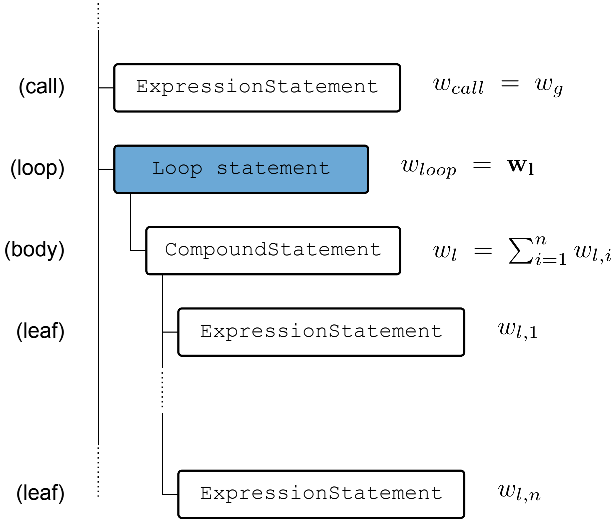
Checkpoint insertion

Cost estimation



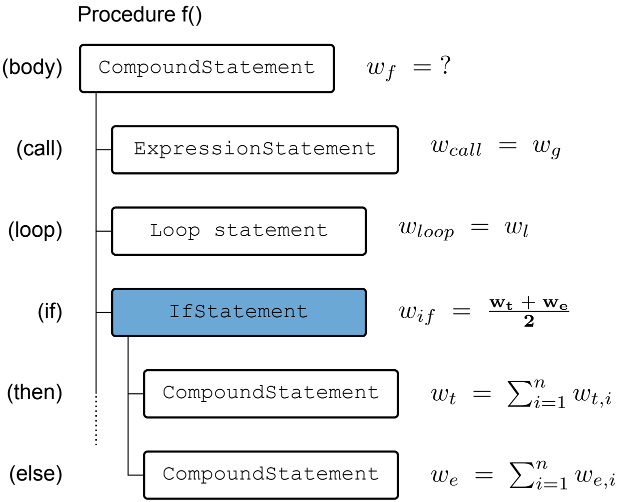
Checkpoint insertion

Cost estimation



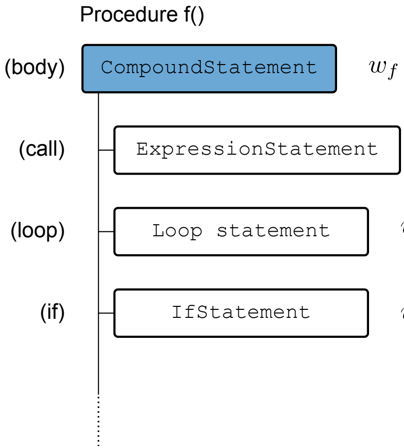
Checkpoint insertion

Cost estimation



Checkpoint insertion

Cost estimation



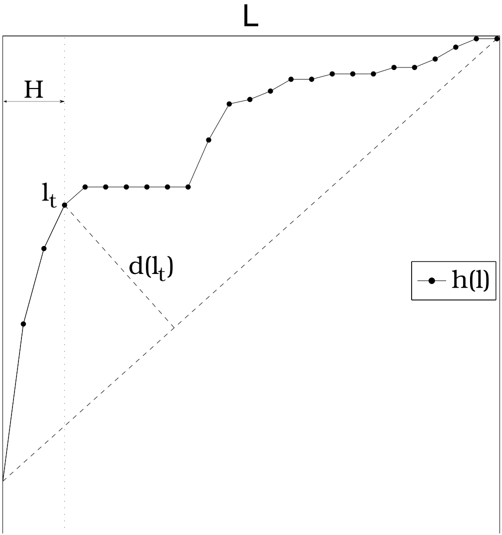
$$w_f = \mathbf{w_{call}} + \mathbf{w_{loop}} + \mathbf{w_{if}} + \ldots$$

$$w_{call} = w_g$$

$$w_{loop} = w_l$$

$$w_{if} = \frac{w_t + w_e}{2}$$

Checkpoint insertion
Loop thresholding

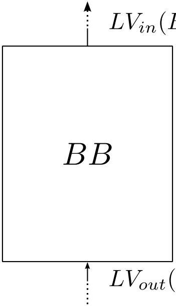


Live variable analysis
Overview

- 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

$$LV_{in}(BB) = USE(BB) \bigcup \{LV_{out}(BB) - DEF(BB)\}$$

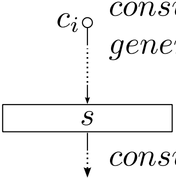


$$LV_{out}(BB) = \bigcup_S LV_{in}(S)$$

$$LV_{out}(BB_{end}) = \emptyset$$

Live variable analysis

Traversing the code

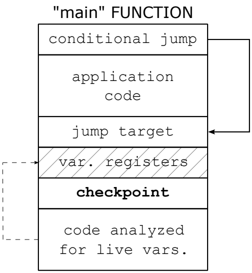


$$consumed = \emptyset$$
$$generated = \emptyset$$

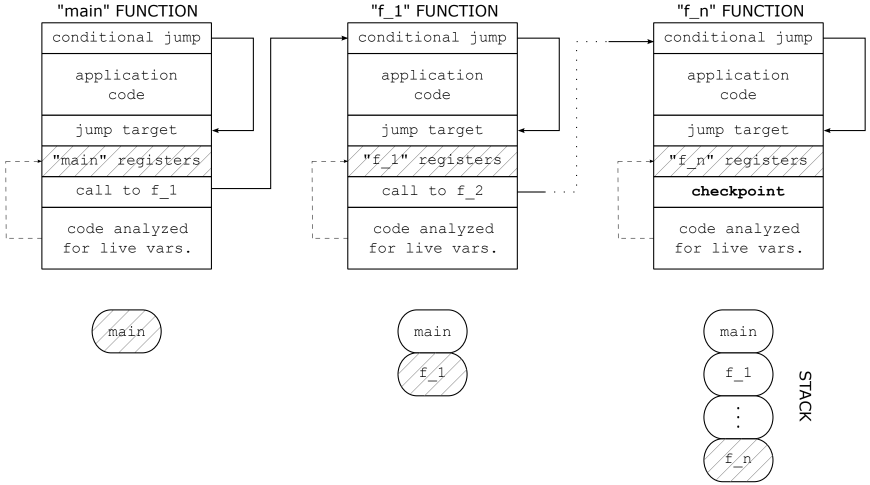
$$consumed = consumed \bigcup \{USE(s) - generated\}$$
$$generated = generated \bigcup DEF(s)$$

$$\mathbf{LV_{in}(c_i) = consumed}$$

Putting it all together



Putting it all together



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`: `**`, `//`.

Perceptions on the Cetus infrastructure

Perceived strengths

- Java implementation: portability and clean design.
- Completely open architecture from head to toe.
- High level representation.
- Evolving infrastructure (e.g. new built-in analyses).

Perceived weaknesses

- Complex IR.
- Performance.

Cetus-assisted checkpointing of parallel codes

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

COMPUTER ARCHITECTURE GROUP
UNIVERSITY OF A CORUÑA
GAC.UDC.ES

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