C^3

the Cornell Checkpoint (pre-)Compiler

Daniel Marques
Department of Computer Science
Cornell University

CS 612
April 10, 2003

Outline

- Introduction and background
- Checkpointing process state
    - Checkpointing a process’ position
    - Checkpointing local and global variables
    - Checkpointing heap objects
- Optimizing checkpoint overhead

Project background

- In the past, the High Performance Computing (HPC) community didn't give much though to fault-tolerance (FT)
  - Expensive, specially designed "monolithic" machines
- Today, the migration to distributed, interconnected machines (made from off-the-shelf components) requires a re-examination of FT needs and strategies

Increasing complexity

- Machines are increasing in complexity
  - Newest ASCI machine @ 30,000 processors
  - IBM’s BlueGene-L has 65k nodes, each with two processing cores

- Increase in the possible points of failure ⇨increase in failure rate
  - Once per hour, or even more frequently
  - Measured as the Mean Time Between Failure (MTBF)

Existing solutions don’t apply

- FT has been extensively studied since the first crash of a machine
- Typically, FT solutions have involved *system-level* checkpointing (SLC)
  - OS or library linked with application intermittently writes the whole state of application to stable storage (core-dump style)
- Works OK for uniprocessors, but with thousands of processes, each using GBs of address space, the overhead of saving all this data to stable (network) storage is way too high

Application-level checkpointing

- Typically, for these massive applications, what works is *application-level* checkpointing (ALC)
  - The programmer augments the source to his application such that it can save and restart from its state
  - Only need to save the minimum amount of state required to resume
    - i.e. for an engineering simulation, just save the physics of the problem, not the computational structures
- Requires extra effort for the programmer, which can not be amortized over many applications
- For some programming models (e.g. without barriers) it can be impossible to determine a point where the processes will by "synched" correctly to save just the physics

Proposal

- Our solution is to use compiler technology to generate ALC automatically for massively parallel computations
  - As easy to use as SLC
  - As efficient as hand-written ALC
- Two separate components
  - Mechanism to save state of an individual process
  - Mechanism to save state of communication channels
- Our first goal is for providing FT for C applications that use the MPI communication library

C^3 : the Cornell Checkpoint (pre-)Compiler

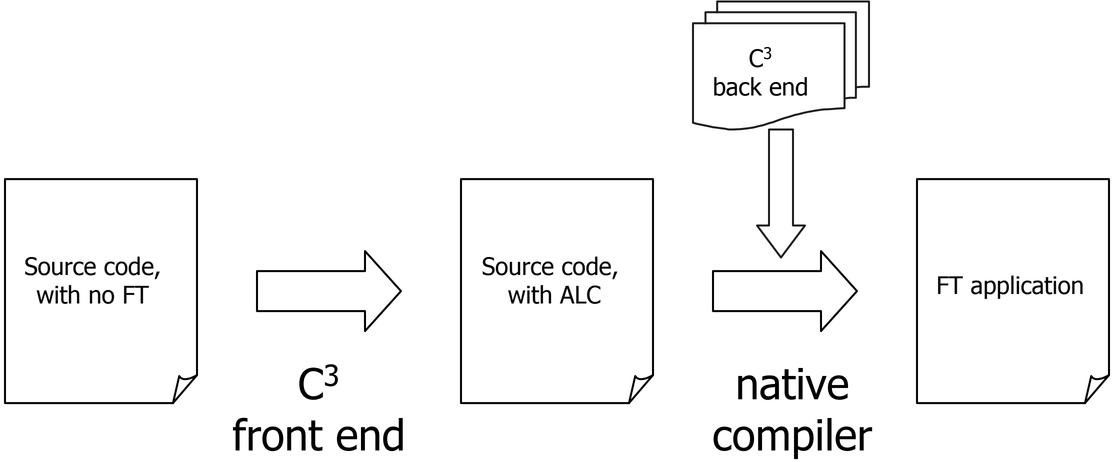
- Today's talk with focus on the Cornell Checkpoint (pre-)Compiler (C^3)

- C^3 is a source-to-source compiler that transparently adds ALC to the source code of a C application

- Actually, it consists of two parts, the front end, and the back end, which is a set of runtime libraries that you link the modified application against
  - The front end inserts the appropriate calls to this library in the application's source

C^3 usage

- C^3 front end will insert code to save / restore a processes state, only at specific points marked in the code
  - *ccc_potential_checkpoint* locations
  - Another analysis could be used to determine good locations
- This modified source will then be passed to the native compiler, which will generate code for a FT application and link it with the C^3 back end libraries



Outline

- Introduction and background
- Checkpointing process state
  - Checkpointing a process’ position
  - Checkpointing local and global variables
  - Checkpointing heap objects
- Optimizing checkpoint overhead

Process state

- The state of a process consists of a set of different elements
    - Its program text
    - Its position in the program text (PC)
    - Its current activation record (stack frames)
    - Its global, local, and heap allocated variables
- All of these need to be saved, and restored correctly for a FT solution to be correct

Process position (1)

- On restart, a process needs to resume at the statement immediately following the *ccc_potential_checkpoint* statement where the checkpoint was taken
- The front end will insert a unique label for every such statement in the program
    - Expands statement first, see example

Example (1)

```
bar()
{
    int x;
    //...
    ccc_potential_checkpoint();

    //...

    ccc_potential_checkpoint();
    //...
}
```

```
bar()
{
    int x;
    //...
    if(checkpoint_time)
    {
        take_checkpoint();
label_1:
    }
    //...
    if(checkpoint_time)
    {
        take_checkpoint();
label_2:
    }
    //...
}
```

Process position (2)

- We need to ensure that we restart at the correct checkpoint location

- The precompiler inserts code to manipulate a data structure, called the *Position Stack* (PS), such that top of the stack always contains the value of the label of the checkpoint we are about to take

- Why a stack to store only one value? You will see

Example (2)

```
bar()
{
    int x;
    //...
    ccc_potential_checkpoint();

    //...

    ccc_potential_checkpoint();
    //...
}
```

```
bar()
{
    int x;
    if(checkpoint_time)
    {
            PS.push(1);
            take_checkpoint();
label_1:
            PS.pop();
    }
    //...
    if(checkpoint_time)
    {
            PS.push(2);
            take_checkpoint();
label_2:
            PS.pop();
    }
    //...
}
```

Process position (3)

- When the checkpoint is taken, the PS is saved as part of the checkpoint

- On restart, we restore the PS, and then use the value(s) stored on it to go to the first statement after the checkpoint was taken

- We do this by having the precompiler insert a jump-table at the entry to the function (after the variable declaration)

Example (3)

```
bar()
{
    int x;
    //...
    ccc_potential_checkpoint();

    //...

    ccc_potential_checkpoint();
    //...
}
```

```
bar()
{
    int x;
    //...
    if(RESTARTING)
    {
          int x = PS.top();
          switch(x)
          {
          case 1: goto label_1;
          case 2: goto label_2;
          }
    }
    //...
}
```

Process position (4)

- It is not enough to just know the checkpoint location we are restoring to, we also need to know the function call chain that got us there
- The front end will also insert labels before each function call, and the appropriate manipulations of the PS

Example (4)

```
foo()
{
    int y;
    //...
    bar();


    //...
    bar();

    //...
}
```

```
foo()
{
    int y;
    //...
    if(RESTARTING)
         switch(PS.top()) {…}
    //...
    PS.push(1);
label_1:
    bar();
    PS.pop();
    //...

    PS.push(2);
label_2:
    bar();
    PS.pop();
    //...
}
```

Process position (5)

- In this manner, when we restore from a checkpoint, the PS contains a record of all the function calls that were made, until we arrived at the take_checkpoint site
- When we restart, execution begins in main(), the RESTARTING flag is set, and each function jumps to the same call it made leading up to the checkpoint
- Finally, we arrive at the deepest function, where we jump to the statement after the checkpoint, the flag is unset, and execution proceeds normally

Caveat: Decomposing complex expressions

- As we saw, each function call must have its own unique label
    - Otherwise, how do we know which function to resume to
- Expressions that contain multiple functions calls must be decomposed into a sequence of smaller expressions

```
z = callA(x++)* callB(callC()++);
```

becomes

```
temp1 = callA(x++);
temp2 = callC()++;
temp3 = callB(temp2);
z = temp1 * temp3;
```

So that we may insert the appropriate code before and after each of the function calls

Outline

- Introduction and background
- Checkpointing process state
    - Checkpointing a process' position
    - Checkpointing local and global variables
    - Checkpointing heap objects
- Optimizing checkpoint overhead

Local variables (1)

- On restart, our program will resume immediately after the most recent checkpoint
  - The activation stack will have the same frames, in the same relative position
- The stack frames have ‘garbage’ values for the stack variables

Local variables (2)

- Force stack variables to have the same virtual address for every run of the same executable (set stack-base appropriately)
- Use another data structure, the *Variable Descriptor Stack* (VDS), to save and restore variable values
- Front end inserts code such that:
    - When a variable enters scope, push its address and length onto VDS
    - When it leaves scope, pop it from the VDS

Example (5)

```
function(int a)
{
   VDS.push(&a, sizeof(a));
   int b[10];
   VDS.push(&b, sizeof(b));
   {
        int c;
        VDS.push(&c, sizeof(c));
        //...
        VDS.pop;
   }
   VDS.pop;
   VDS.pop;
}
```

Local variables (3)

- When we take a checkpoint we use the VDS to copy the variables’ values from the stack, into the checkpoint file
- We also save the VDS as part of the checkpoint
- On restart, we first restore the stack, then restore the VDS, and then use it to copy values from the checkpoint file into the variables’ locations

Example (6)

```
Save_variables()
{
    int j;
    int x = VDS.length();
    for(j = 0; j < x; j++)
    {
          item = VDS.item(j);
          ad = item.address;
          size = item.size;
          fwrite(ckpt_file,
                    ad, size);
    }
    Save(VDS);
}
```

```
Restore_variables()
{
    int j;
    int x;
    Restore(VDS);
    x = VDS.length();
    for(j = 0; j < x; j++)
    {
          item = VDS.item(j);
          ad = item.address;
          size = item.size;
          fread(ckpt_file,
                    ad, size);
    }
}
```

Return statement

- When we encounter a return statement, we must pop all variables currently in scope
    - Those declared in all enclosing scopes, up to the function declaration

Caveat: nested scopes

- With nested scopes, we actually would need to maintain more than one entry on the PS for each function
  - So that we could jump to variable declarations in nested scopes
- We avoid this by moving all variables to the function level
  - Doing appropriate renaming

Global variables

- Currently, we treat global variables as local variables to a *pre-main()* function that is called before main
- Accomplished by having the front end rename main() to usr_main(), and insert code for a new main() function that manipulates the VDS for the globals, and calls usr_main()
- Doesn’t work for multiple files where the global are not all extern-ed

Outline

- Introduction and background
- Checkpointing process state
  - Checkpointing a process’ position
  - Checkpointing local and global variables
  - Checkpointing heap objects
- Optimizing checkpoint overhead

Pointers

- Notice that we saved stack variables regardless of what they held
- This means, that if a variable held a valid pointer, on restart it will be restored with that same value
- So the same object must be restored to the same address
- For pointers to the stack, this is accomplished by always building the stack in the same manner from the same starting address

Heap objects (1)

- For objects on the heap, this is not so easy (malloc provides us with no way to specify an address)
- We also need to ensure that the heap’s control data (the free list, etc.) are restored correctly, so that future calls to malloc, free, etc. work as expected
- We need to build and maintain our own heap!

Heap objects (2)

- Use operating system calls to request a block of memory at a specific address
  - For Windows NT – VirtualAlloc
- Build a heap in that area
- We could just use a large global array, but might be less efficient
  - Maybe for porting to other OS where no analogue is available

Heap objects (3)

- Back end provides our version of the memory management routines
- Front end converts calls to malloc, etc., to our versions (ccc_malloc) in the application source

Heap objects (4)

- In addition to a free-list, we also maintain a used-list
- Useful for checkpointing, so that we only save heap objects that are not free
  - Trade off between saving less data and the overhead of list traversal / cache misses
  - Might adjust dynamically at runtime, dependent on heap "fullness"
- At checkpoint time, either save entire heap, or only the non-free items. Also save free-list and used-list
- On restart, request same area of memory, restore the free-list and used-list, and then copy items from checkpoint onto heap, to their original address

Revisiting pointers

- Because both stack variable and heap objects are restored to the same addresses as they originally had, we need to make no special consideration regarding any pointers
- We save them as ordinary data

Another approach

- The PORCH system (Ramkumar, Strumpen (Iowa / MIT)) is another approach to compiler inserted checkpointing
  - Goal was portability, i.e. re-locatable pointers
  - Requires using a subset of language
  - And meta-structures to describe “links” in data structures

Outline

- Introduction and background
- Checkpointing process state
  - Checkpointing a process’ position
  - Checkpointing local and global variables
  - Checkpointing heap objects
- Optimizing checkpoint overhead

Memory exclusion (1)

- As described so far, a checkpoint contains all of a program's data, but there are situations where we do not need to save it all
  - Dead data – memory holding a value that will not be read again
  - Static data – memory that hasn't changed since the previous checkpoint
  - Recomputable data – memory that holds a value that can be recomputed after restoration
  - Redundant data – memory that holds a value that is also stored someplace else

Memory Exclusion (2)

Dead memory

```
{
  a = 7;
  //…
  checkpoint();
  // if there are no
  // reads to a in this
  // region, we don’t
  // need to save it
  // above
  a = 9;
}
```

Static memory

```
{
  a = 7;
  //…
  checkpoint();
  // if there are no
  // writes to a in
  // this region, we
  // don’t need to save
  // it below
  checkpoint();
}
```

Memory exclusion (3)

- Fairly easy compiler analysis to determine when a stack variable is dead or static
- Does not work for heap objects
  - Objects are anonymous
  - Objects might have multiple aliases
- Compiler analysis becomes very difficult
  - Must be sound / conservative
- We must use dynamic systems to do exclusion
  - But use compiler to “guide” them

Dead object elimination

- Use a (conservative) garbage collector to eliminate garbage before a checkpoint
  - Free garbage
- Use compiler analysis to determine if a stack allocated pointer is dead
- Pass that information to the GC
  - i.e. the GC will ignore the fact that x points to some object when determining if it is garbage

Static object elimination

- Incremental checkpointing – use page protection mechanism to only checkpoint pages that have changed since the last checkpoint
- Suffers from the false-write problem
  - All data on a page will get checkpointed, if anything on the page has changed
- Use compiler to determine which objects should be allocated next to one another

Recomputable object elimination

- The value of a certain object (result objects) may be a function of the values of other objects (operand objects)
- Rather than save all the objects, just save the operand objects
- On restart, recompute the result object
- API can specify what could be recomputed, and how to do it
- Use a compiler to discover this automatically

Example (7)

```
while()
{
        // B and C are vectors
        A = B × C; //cross product
        ignore(A);
        checkpoint;
Label_1:
        if(restart)
                A = B × C;
}
```

Redundant object elimination

- An MPI application consists of independent processes, each with its own address space
- A particular value might be stored on multiple nodes
  - In fact, MPI has many functions to cause such behavior, MPI_Broadcast, etc.
- Only save data on one node, on restart, send it to all others
- Again, API to specify such objects
- Use compiler analysis to automate this

Checkpoint location

- The location of checkpoints can have a drastic effect on performance
  - If we test to see if we need to take a checkpoint too frequently, additional overhead
  - Might be able to eliminate more memory at different locations
- Currently, programmer must specify checkpoint locations in the code
  - By a call to ccc_potential_checkpoint()
- Use compiler to determine optimal locations