Portable Checkpointing for Heterogeneous Architectures

Balkrishna Ramkumar
Dept. of Electrical and Computer Engineering
University of Iowa
Iowa City, IA 52242
`ramkumar@eng.uiowa.edu`
Phone: 319-335-5957

Volker Strumpen
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
`strumpen@theory.lcs.mit.edu`
Phone: 617-253-1531

Abstract

Current approaches for checkpointing assume system homogeneity, where checkpointing and recovery are both performed on the same processor architecture and operating system configuration. Sometimes it is desirable or necessary to recover a failed computation on a different processor architecture. For such situations checkpointing and recovery must be *portable*. In this paper, we argue that source-to-source compilation is an appropriate concept for this purpose. We describe the compilation techniques that we developed for the design of the `c2ftc` prototype. The `c2ftc` compiler enables machine-independent checkpoints by automatic generation of checkpointing and recovery code. Sequential C programs are compiled into fault tolerant C programs, whose checkpoints can be migrated across heterogeneous networks, and restarted on binary incompatible architectures. Experimental results on several systems provide evidence that the performance penalty of portable checkpointing is negligible for realistic checkpointing frequencies.

1 Introduction

Large distributed systems are inherently heterogeneous in nature. Even relatively small local area networks usually consist of a mixture of binary incompatible hardware components and are operated by an even larger variety of operating systems. Providing application fault tolerance in such environments is a key technical challenge, especially since it requires that checkpointing and recovery be *portable* across the constituent architectures and operating systems.

Obstacles to portability range from the architecture over the operating system to the language level. *Architectural hazards* include differences in representations of basic data types and alignments, or specialized hardware support for programming languages, e.g. register windows of SPARC architectures, that affect the stack frame layout. Differences among implementations of *operating systems*, such as UNIX flavors BSD/System V/Linux, surface in specific process address space layouts and system calls. Several *programming language features* are not portable. Dynamic memory management schemes vary from system to system. Pointers are not portable across different address space layouts. Implementations of low-level features like `setjmp/longjmp` and socket or pipe-based communication mechanisms further exacerbate the problem.

In this paper, we argue that the two key requirements for portable checkpoints — stack environment portability and pointer portability — can be provided by means of source-to-source compilation based on the following code transformations: (1) Provide machine-independence of the stack environment by compiling entry points into functions that resemble computed goto's. The key to manipulating the program counter and stack pointer in a portable manner is to use the basic function call and return mechanism at the source code level. (2) Pointers are transformed into portable, checkpoint internal offsets. Other aspects such as data representation conversion, and runtime support are not covered in this article, but can be found in [9].

We have developed `c2ftc`, a source-to-source compiler that translates C programs into fault tolerant C programs. This prototype generates code for saving and recovering portable checkpoints to enable fault tolerance across heterogeneous architectures. The `c2ftc` compiler instruments the source program based on potential checkpoint locations in the program specified by the programmer. These checkpoints can be restored on binary incompatible architectures. `c2ftc` maintains checkpoints in a *Universal Checkpoint Format* (UCF), a machine independent format which is customizable for any given configuration of heterogeneous machines by specifying basic data types including byte order, size and alignment. Other repesentation issues such as the encoding of denormalized numbers can be handled by supplying architecture specific conversion routines.

The remainder of this paper is organized as follows. In Section 2, we describe related work in the area of checkpointing and recovery. Section 3 describes the benefits of the source-to-source compilation approach to portable checkpointing. Our algorithm for checkpointing the runtime stack in the presence of pointers is described in some detail in Section 4. In Section 5, we present experimental results showing that the performance penalty for portability is negligible. Finally, we discuss limitations of compiler support for portable checkpointing in Section 6.

2 Related Work

Elnozahy *et al.* [2] and Plank *et al.* [6] have proposed efficient implementation techniques to minimize the overhead of checkpointing to few percent of the execution time. The techniques developed in [2, 6] rely on efficient page-based bulk copying and hardware support to identify memory pages modified since the last checkpoint. Unfortunately, these optimizations are restricted to binary compatible hardware and operating systems. However, the remarkably low overhead afforded by these optimizations suggests that it is possible to achieve checkpoint portability at only marginally higher cost.

The issue of portability across heterogeneous architectures has been addressed in the language community [3, 4]. Languages like Java [4] provide an interpreter-based approach to portability where the program byte code is first "migrated" to the client platform for local interpretation. Unfortunately, such schemes severely compromise performance since they run at least an order of magnitude slower than comparable C programs. Another possibility is "compilation on the fly" [3] which provides portability by compiling the source code on the desired target machine immediately prior to execution. This technique requires the construction of a complex language environment. Moreover, to date neither interpreter-based systems nor compilation on the fly are explicitly designed to support fault tolerance.

Li and Fuchs [5] were among the first to demonstrate the use of compilers for automatically inserting potential checkpoint locations into programs. At runtime, heuristics are used to determine which of these checkpoints will be activated, with the goal to minimize the checkpointing overhead. This work does not address portability.

Theimer and Hayes [10] present a recompilation-based approach to heterogeneous process migration. The idea to utilize compilation for that purpose is similar to our approach. However, the compilation technique proposed is very different. Their idea is to, upon migration, translate the state of a program into a machine independent state. Then, a migration program is generated that represents the state, and can be compiled on a target machine. When run, the machine independent migration program recreates the process. Rather than compiling a migration program each time that a checkpoint is to be taken, we instrument the original program with code that barely affects the runtime during normal execution, avoids the overhead of compiling a migration program, and is conceptually much simpler.

Zhou *et al.* [11] describe the Mermaid system for distributed shared memory on heterogeneous systems. This system is not fault tolerant, but generates data representation conversion routines automatically for all shared memory objects. This paper provides a detailed treatment on conversion. A major difference to `c2ftc` is the conversion code generation for complex data types. Wheres Mermaid uses "utility software" to generate this code, `c2ftc` utilizes the information provided by the abstract syntax tree to this end. Another design decision of Mermaid is to dedicate a page of memory to a particular data type. Although the authors defend this scheme in the context of dynamically allocated shared memory, such an organization is clearly impractical for the runtime stack, which has to be converted too when saving a checkpoint.

Seligman and Beguelin [7] have developed checkpointing and restart algorithms in the context of the Dome C++ environment. Dome's checkpointing is designed for portability, but requires that the program be written in the form of a main loop that computes and checkpoints alternately. This obviates the need to store the runtime stack. In contrast, our approach provides a general mechanism to save the runtime stack.

3 Source-to-Source Compilation

We argue that source-to-source compilation provides an elegant solution for portable checkpointing in heterogeneous environments. The design of the `c2ftc` compiler has shown that source-to-source compilation solves the following three key problems:

- Preservation of program semantics: Source-to-source code transformations guarantee that program semantics remain invariant, because they are target independent.

- Conversion of program data: Type information is in general not available at runtime. Generation of conversion code at compile time ensures its availability during checkpointing.

- Portable runtime support: The approach permits the choice of transformations for fault tolerance that do not expose architecture dependencies, thereby enabling portable runtime support.

Source-code level transformations can provide portability where system-based approaches would become very complex or even fail. For example, in order to capture the state of the stack at the system level, not only do program counter, stack pointer and other architecture specific state need to be saved, knowledge of compiler-specific behavior is also necessary. The stack frame layout must be known in order to associate type information with corresponding memory locations to effect conversion of data representations. Some compilers do not reserve stack space for variables stored in registers. In such cases, a system-based approach would also have to provide support for saving and restoring register contents across machines with potentially different register sets. It is unclear how portability could be provided in such situations.

Source-to-source compilation also provides opportunities for additional optimizations that cannot be exploited by system-based schemes:

- It is possible to perform live variable analysis at compile time and reduce the amount of data that needs to be checkpointed.
- Compile-time analysis can be used to identify potential checkpoint locations in a program to the end of reducing checkpointing overhead.
- Compile-time analysis can be used to support garbage collection of the heap at run time before checkpoints are taken.

However, it remains to be seen whether compile-time analysis is powerful enough to reduce checkpointing overheads to the degree that page-based techniques as proposed by Elnozahy *et al.* [2] improve performance.

Source-to-source compilation does furthermore permit the use of system-level optimizations like *copy-on-write* or communication latency-hiding [8] during the transfer of a checkpoint to stable storage. This benefit is significant, since this transfer represents the most time-consuming part of checkpointing, given today's relatively low network and disk performance.

4 Shadow Checkpointing

`c2ftc` applies fault tolerance transformations to the abstract syntax tree of a C program. These transformations involve analysis, minor changes to the source code such as moving function calls out of expressions, and adds new code to effect portable checkpoints at the application level.

Currently, the user must specify potential checkpoint locations by inserting a call to the library function `checkpoint`. The frequency of checkpointing is controlled using a timer that activates checkpointing when the next potential checkpoint location is visited. Then, the state of the program is pushed onto the *shadow stack*, which is subsequently saved on stable storage. The shadow stack is maintained in the Universal Checkpoint Format. On UCF-incompatible machines, data are converted on-the-fly while pushing the state onto the shadow stack, and vice versa during recovery. The code for pushing and popping variables from the shadow stack as well as for conversion is compiler generated.

The shadow stack essentially doubles the memory requirement of an application. One of several options to service memory, if the DRAM cannot hold the shadow stack, is to memory-map the shadow stack to local disk, trading in checkpointing overhead for memory requirement. This will still be substantially faster than transferring a large checkpoint via a network.

The Universal Checkpoint Format specifies the layout of a portable checkpoint by specifying the data representations and alignments of basic data types. UCF is a flexible and adaptable format that can be customized to a particular network by specifying byte order, size and alignment of basic types, as well as complex data representations such as denormalized numbers. Typically, data representations and alignments of the majority of available machines in the network should be chosen as the UCF format to minimize the overhead of converting data types to and from the UCF format on UCF-incompatible systems. In evolving networks, the UCF format can be changed as frequently as necessary; this only requires that programs requiring checkpointing be recompiled before execution.

Several UCF-related issues involving the conversion of pointers into offsets, the transparent generation of conversion code for complex data types, the handling of function pointers, pointers from stack to heap, heap to stack, data/bss to stack, stack to data/bss, etc., are not presented here due to lack of space. A discussion may be found in [9].

4.1 Checkpointing the Stack

We begin by first considering only non-pointer variables on the runtime stack. The algorithm is then extended in Section 4.2 to support pointer variables.

The basic approach for saving the variables on the stack is to visit each stack frame, and save its local variables identified at compile time.[1] The stack is checkpointed by returning the active function call sequence, thereby visiting each individual stack frame starting from the top of the stack down to the bottom. For each stack frame visited, the state of the local variables is pushed onto the shadow stack. The stack must then be restored by executing the original function call sequence again. `c2ftc` generates code to access each local variable by name rather than block-copying the stack. This eliminates problems caused by non-portable implementations based on setjmp/longjmp pairs, as for example used in `libckpt` [6].

In order to preserve the program's state while checkpointing, none of the program's statements may be executed. Therefore, the program must be instrumented with function calls and returns to visit the each stack frame during checkpointing without affecting the semantics of normal execution. Compile time analysis identifies function call sequences that can lead to a potential checkpoint location. All functions that lie in such a sequence are subject to instrumentation.

For each function requiring instrumentation, stack *growth* may happen in one of two modes: *normal execution*, or *stack restoration*. For the latter, it is necessary to supply a "computed goto" at the top of the function body that causes a jump to the next function call in the call sequence leading to the checkpoint location. This is accomplished by `c2ftc` by inserting a jump table with `goto` statements to each of the function calls in the function body that can lead to a potential checkpoint location.

Stack *shrinkage* may also occur in one of two modes: *normal execution*, or *stack saving* when an activated checkpoint location is visited. For the latter, it is necessary to provide a function call wrapper that will save variables in the current stack frame upon return from the function call, and then cause a return from the calling function to save its parent's frame.

We illustrate the transformations performed by `c2ftc` by means of an example. Figure 1 shows a recursive program to compute Fibonacci numbers. Functions `main` and `checkpoint` are provided in a library. Here, `main` is supplied only to clarify the function call sequence. The application consists of the functions `chkpt_main`, which substitutes the original function `main` by renaming, and function `fib`. We assume that a potential checkpoint location is specified within `fib` by means of a call to function `checkpoint`. `c2ftc` transforms function `fib` into the code spread over Figures 2, 3 and 4. Function `main` is transformed analogously.

The program may execute in one of four modes.

[1] An optimized version would only save the live variables determined by data-flow analysis.

```
extern int checkpoint();

main(int argc, char *argv[]) {
  chkpt_main(argc, argv);
}

chkpt_main(int argc, char *argv[]) {
   int n, result;

   n = atoi(argv[1]);
   result = fib(n);
}

fib(int n) {
   if (n > 2)
      return (fib(n-1) + fib(n-2));
   else {
      checkpoint();
      return 1;
   }
}
```

Figure 1: Code fragment to illustrate the Shadow Checkpoint Algorithm.

This mode of execution is kept in the global state variable `_SL_chkptmode`.

Normal execution: During normal execution of the program the execution mode is set to `_SL_EXEC`. The jump table is skipped (Figure 2), and variable `_SL_callid` is assigned to encode the entry point into the function for use during the restore and recover phases (Figure 3).

Save phase: The variables of the stack frames are saved on the shadow stack. Value `_SL_SAVE` is assigned to `_SL_chkptmode` in function `checkpoint` before it returns. Then, the variables of the calling function are stored, and this function returns. This process is repeated until all stack frames on the call sequence between `main` and `checkpoint` are popped from the runtime stack. Local variables, including `_SL_callid`, are saved by macro `_SL_SAVE_fib_0` given in Figure 4.

Restore phase: The runtime stack, which has been destructed during the save phase, is reconstructed during the restore phase by reexecuting the original call sequence from `main` to `checkpoint`. Value `_SL_RESTORE` is assigned to `_SL_chkptmode` in function `main`. Since more than one function call may lie on a call sequence to `checkpoint`, variable `_SL_callid` is used to identify which call is in the call sequence being restored, cf. Figure 3. Local variables are restored by macro `_SL_RESTORE_fib_0` given in Figure 4.

Recover phase: Recovery is almost the same as the restore phase. The only difference is that the vari-

```
int fib(int n)
{
   unsigned long _SL_callid, _SL_addr;
   int _SL_fun0, _SL_fun1;

   switch (_SL_chkptmode) {
     case(_SL_EXEC): break;
     case(_SL_RESTORE):
       _SL_addr = s_stack.top;
       _SL_RESTORE_fib_0;
       _SL_CONVERT_fib_0(_SL_addr);
       switch(_SL_callid) {
         case(0): goto L_SL_call0;
         case(1): goto L_SL_call1;
         case(2): goto L_SL_call2;
       }
     case(_SL_RECOVER):
       _SL_addr = s_stack.top;
       _SL_CONVERT_fib_0(_SL_addr);
       _SL_RESTORE_fib_0;
       switch(_SL_callid) {
         case(0): goto L_SL_call0;
         case(1): goto L_SL_call1;
         case(2): goto L_SL_call2;
       }
   }
         :
         :
```

Figure 2: The *jump table* generated at the entry of function `fib`.

```
          :
          :
    if (n > 2) {
      _SL_callid = 0;
L_SL_call0:
      _SL_fun0  = fib(n-1);
      switch (_SL_chkptmode) {
        case(_SL_EXEC): break;
        case(_SL_SAVE): _SL_SAVE_fib_0;
           return 0;
      }

      _SL_callid = 1;
L_SL_call1:
      _SL_fun1 = fib(n-2);
      switch (_SL_chkptmode) {
        case(_SL_EXEC): break;
        case(_SL_SAVE): _SL_SAVE_fib_0;
                        return 0;
      }
      return _SL_fun0 + _SL_fun1;
    }
    else {
      _SL_callid = 2;
L_SL_call2:
      checkpoint();
      switch (_SL_chkptmode) {
        case(_SL_EXEC): break;
        case(_SL_SAVE): _SL_SAVE_fib_0;
                        return 0;
      }
      return 1;
    }
}
```

Figure 3: The *function call wrappers* generated in the body of function `fib`.

ables have to be converted before they can be popped from the shadow stack, whereas during the restore phase they need to be restored, and then converted to be available in UCF representation on the shadow stack, cf. Figure 2. The conversion function `_SL_CONVERT_fib_0` is shown in Figure 4.

Note that all variables on the runtime stack are accessed by name to push and pop them from the shadow stack (Figure 4). This renders the checkpointing code independent of differences in the organization of the runtime stack on different machines. Once the state of the runtime stack has been restored, the contents of the shadow stack is part of the checkpoint, which can be written to stable storage.

4.2 Pointers

The basic idea to provide pointer portability is straightforward: Pointers are translated into displacements within the checkpoint. The implementation of this idea is described in the following.

We classify pointers using two orthogonal categories: their *target segments* and the *direction* denoting the order in which the pointer and its target are pushed onto the shadow stack. The following *target segments* are common in UNIX environments, and have to be distinguished when treating pointers since segment addresses and sizes differ from target to target. In the UCF format, all pointer displacements are tagged to identify their target segments.

1. **Stack pointer:** The shadow stack offset is the displacement between the pointer address on the shadow stack and its target on the shadow stack.

2. **Heap pointer:** The shadow stack offset is calculated with respect to the bottom of the heap segment. The use of user-level memory management ensures that this offset is target invariant.

3. **Data/bss pointer:** The shadow stack offset is the displacement between the pointer address on the shadow stack and its target on the shadow stack.

4. **Text pointer:** These are function pointers or pointers to constant character strings in C. The latter do not require any special attention, because they will be available automatically after recovery. Function pointers are translated into a unique identifier assigned by the runtime system.

```
#define _SL_SAVE_fib_0 { \
  *--((unsigned long *) s_stack.top) = _SL_callid; \
  *--((int *) s_stack.top) = n; \
  *--((int *) s_stack.top) = _SL_fun0; \
  *--((int *) s_stack.top) = _SL_fun1; \
}

#define _SL_RESTORE_fib_0 { \
  _SL_fun1 = *((int *) s_stack.top)++; \
  _SL_fun0 = *((int *) s_stack.top)++; \
  n = *((int *) s_stack.top)++; \
  _SL_callid = *((unsigned long *) s_stack.top)++; \
}

static __inline__ void _SL_CONVERT_fib_0(addr)
unsigned long addr;
{
  _SL_conv_word(addr);   ((int *) addr)++;
  _SL_conv_word(addr);   ((int *) addr)++;
  _SL_conv_word(addr);   ((int *) addr)++;
  _SL_conv_word(addr);   ((unsigned long *) addr)++;
}
```

Figure 4: Compiler generated code for saving, restoring and converting the variables in function `fib`.

Pointers with these four targets can exist as automatic variables on the stack, dynamically allocated variables on the heap, and as global variables in the data/bss segment. Note that the classification of pointers by their target segments permits the handling of pointer casting or the use of opaque pointers (e.g. `void *`) during parameter passing.

Pointers are also classified with respect to their *direction* relative to the order in which they are pushed onto the shadow stack:

1. **Forward pointer:** The pointer is pushed onto the shadow stack *before* its target object.

2. **Backward pointer:** The pointer is pushed onto the shadow stack *after* its target object.

Call-by-reference parameters are pointers into an ancestor frame on the runtime stack. During execution, the stack frame (callee frame) containing a pointer passed as a parameter is always pushed onto the runtime stack after the caller's frame. During the save phase, the callee frame is pushed onto the shadow stack before the caller frame. Thus, all inter-frame pointers are forward stack pointers. Intra-frame pointers, on the other hand, may be either forward or backward stack pointers.

4.3 Stack Pointers

Forward and backward stack pointers must be treated differently when translating them into machine independent offsets. We consider each of them separately.

4.3.1 Checkpointing Forward Stack Pointers

The conversion of a pointer into its portable offset, when it is saved on the shadow stack, is accomplished by introducing a temporary data structure called a *pointer stack*. The pointer stack keeps track of all pointers found on the runtime stack in order to effect its conversion into its corresponding offset. During the save phase, when a pointer is encountered, two actions are taken: (a) the pointer is copied onto the shadow stack, (b) its shadow stack address is pushed onto the pointer stack. This is necessary, because the location of the target on the shadow stack is not known yet.

During the restore phase, any object being restored to the runtime stack may potentially be the target of one or more pointers elsewhere on the runtime stack. When an object o is restored from address A_o on the shadow stack, entries in the pointer stack are checked to see if the object is a pointer target. If so, for each such pointer on the pointer stack, the difference between the pointer's shadow stack address and the target shadow stack address A_o is computed, and stored in the corresponding pointer stack entry.

Once the entire runtime stack has been restored, the computed displacements in the pointer stack are then written into the corresponding locations in the shadow stack, thereby overwriting the pointer target addresses with portable offsets.

```
extern int checkpoint();

chkpt_main()                function1(long *p)
{                           {
   long a[4];                  p += 1;
                               checkpoint();
   function1(a);               *p = 2;
}                           }
```

Figure 5: Code fragment illustrating the Shadow Checkpoint Algorithm with call-by-reference; cf. Figure 6.

As an example, consider the code fragment in Figure 5 and the illustration in Figure 6. During the save phase, the variables of `function1`, in particular pointer `p`, are pushed onto the shadow stack. In Figure 6, `p` is stored on the stack at X_p, and pushed into X_{ps} on the shadow stack. At this time, a pointer to `p`'s address on the shadow stack X_{ps} is pushed on the pointer stack. Next, the frame of `chkpt_main` is pushed onto the shadow stack. In Figure 6, the target address of `p` is the array element `a[1]`, marked X, and its shadow X_s.

During the restore phase, the frame of `chkpt_main` is restored before the frame of `function1`. Before restoring array `a`, the pointer stack is checked for a reference into `a` on the stack. In this example, the

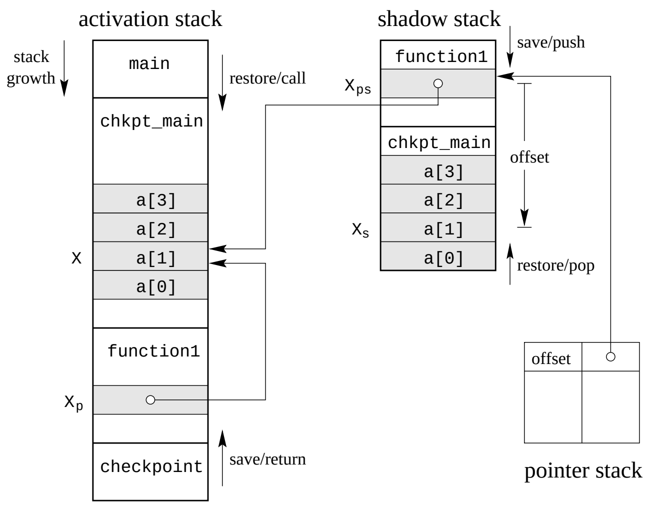

Figure 6: Checkpointing the stack in the presence of a call-by-reference.

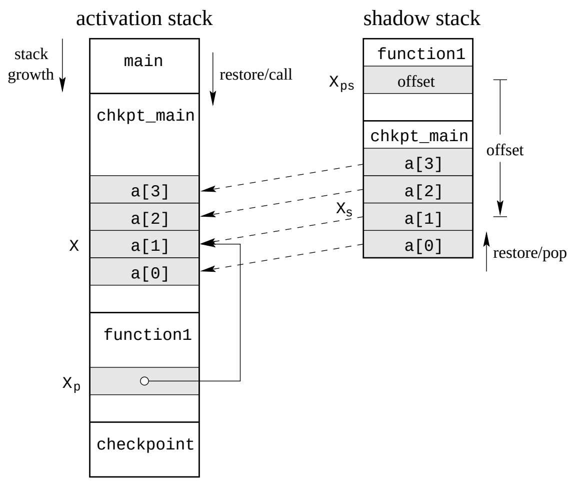

Figure 7: Recovery of the stack in the presence of a call-by-reference.

pointer in X_{ps} points to address X. Note that for arrays it is necessary to check that X_{ps} lies *within the address range* of the array `a`. The shadow stack offset can be computed according to the rule

$$\text{offset} = \text{pointer target address} - \text{pointer address},$$

where both addresses are shadow stack addresses. In Figure 6, offset $= X_s - X_{ps}$. X_{ps} is retrieved from the pointer stack.[2] The offset cannot be stored immediately in X_{ps}, because it holds the value of pointer `p`, which is needed, when restoring the stack frame of `function1`. Once the entire stack is restored, a sweep through the pointer stack copies the offsets into the addresses on the shadow stack. Offset $X_s - X_{ps}$ will overwrite the value of `p` in address X_{ps}.

4.3.2 Recovery of Forward Stack Pointers

Although recovery from a checkpoint is conceptually very similar to the restore phase, recovering pointers presents a difference. All pointer offsets have to be transformed into virtual addresses again. Unlike the checkpointing transformation, this reverse transformation does not require a pointer stack. Figure 7 illustrates the recovery from the checkpoint in Figure 6.

Analogous to the restore phase, the shadow stack is restored from the top to the bottom, i.e. the frame of function `chkpt_main` is copied first. Note that a shadow stack pop operation affects an entire object. Array a is restored as a whole, not element-wise. In order to recover forward pointers — here `p` to $a[1]$ — the address of each object's element on the runtime stack is stored in its location on the shadow stack after the value of the element has been restored on the runtime stack; cf. broken lines in Figure 7. This mapping is needed, when `function1` is restored. The frame of `function1` contains the offset to $a[1]$ in address X_{ps}. Recovering pointer `p` involves the transformation of the offset into the pointer. This requires the lookup operation: `p`$= [X_{ps} + [X_{ps}]]$. The pointer can be found in the shadow stack address which is computed according to the rule:

$$\text{pointer address} = \text{shadow pointer address} + \text{offset}.$$

This simple lookup is bought by saving the complete mapping of the restore target addresses on the runtime stack in the shadow stack. This expense is justified by the fact, that recovery will be the infrequent case.

4.3.3 Backward Stack Pointers

The only backward pointers that might occur on the stack are intra-frame pointers. The number of backward stack pointers can be restricted to the case where the pointer target is another pointer by choosing the order in which variables are pushed on the shadow stack appropriately. `c2ftc` generates save and restore macros such that all non-pointer variables are saved after, and restored before, pointer variables. All pointers to non-pointer variables will then be forward pointers. Only a pointer pointing to another pointer may potentially be a backward stack pointer.

Checkpointing of backward pointers is illustrated in Figure 8, where X_p is a backward stack pointer to X. To deal with backward pointers, the save algorithm presented thus far is modified as follows: For

[2] Determining X_s requires some additional offset computation, details can be found in [9].

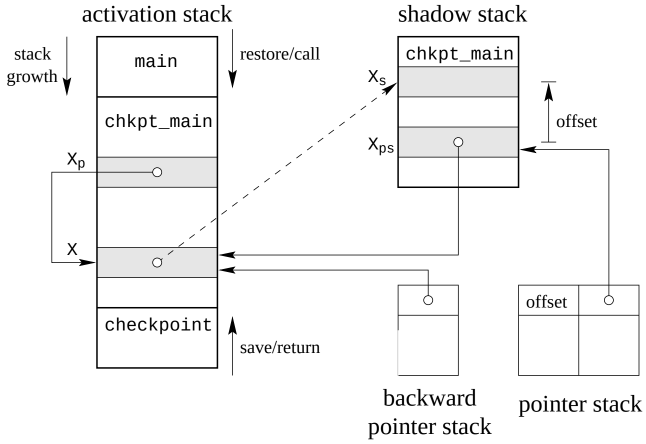

Figure 8: Checkpointing in the presence of a backward stack pointer.

each stack frame, before saving any variables on the shadow stack, all pointer targets of backward pointers are saved in a data structure called the *backward pointer stack*. In Figure 8, X, the pointer target of backward stack pointer X_p is pushed onto the backward pointer stack.

Objects are then copied onto the shadow stack as before. If the object is pointed to from the backward pointer stack, i.e. it is a backward pointer target, its address on the shadow stack is saved temporarily by overwriting the object on the runtime stack with its address on the shadow stack. In the example, the value of X becomes X_s. Next, when the backward pointer in X_p is saved, its shadow address X_{ps} is pushed onto the pointer stack. Furthermore, since the pointer can be recognized as a backward stack pointer by comparing its target address with its own address, the offset is calculated, and saved on the pointer stack. In the example, the offset is $[X_p] - X_{ps} = X_s - X_{ps}$.

The restore phase is the same as before except that it requires the additional step of restoring backward pointers from the backward pointer stack to the runtime stack. Finally, the pointer stack offsets are transferred to the shadow stack as described earlier. Recovery of backward pointers can be implemented similarily to that of forward pointers. However, the pointer stack is needed to store the pointer's shadow address until the target is visited.

The difference in the treatment of forward and backward stack pointers is the computation of the offset. Whereas the offset of forward pointers is computed during the restore phase, offsets of backward pointers can be computed during the save phase, because the pointer target has been copied before the backward pointer is visited.

5 Experimental Results

We present two experiments to evaluate the performance of portable checkpoints: (1) A micro-benchmark measures the code instrumentation penalty, and (2) a small application program demonstrates the checkpointing overhead and performance in the presence of failure and recovery. For more experimental analysis, the reader is refered to [9].

5.1 Code Instrumentation Penalty

The transformation of the Fibonacci program in Figure 1 into the code in Figures 2, 3 and 4 results in a good test case for the runtime overhead due to code instrumentation. The extremely fine granularity of function `fib` yields a program to measure the destruction and reconstruction of small stack frames corresponding to the save and restore phases, whenever the base case of the recursion is visited.

System	plain [s]	instr. [s]	ovh [%]
HP9000/705 / HPUX9.0	9.0	34.9	289
HP9000/715 / HPUX9.0	2.7	11.1	301
i486DX475 / Linux	20.0	38.0	90
SPARCstation1+ / Sunos4.1	27.5	66.7	143
SPARCstation20 / Sunos5.3	5.8	14.5	150

Table 1: Overhead of code instrumentation.

Figure 1 shows measurements of `fib(35)` without storing checkpoints, but executing the save and restore phases of the Shadow Checkpoint Algorithm. Not surprisingly, code instrumentation generates substantial overhead for the Fibonacci program. The cost of a function call increases by a factor 2–4 depending on the architecture. Since this example represents the pathological case where each function call represents an insignificant amount of computation, it provides an empirical upper bound on the runtime penalty paid by the instrumentation.

5.2 Heat Equation

We use a Jacobi-type iteration to solve the heat diffusion problem on a 256×256 grid, executing $1,000$ iterations. Two dynamically allocated two-dimensional `double` arrays are used, one to hold the temperature values of the current iteration, the other to store the results of the five-point-stencil computation. The arrays determine the checkpoint size to be slightly larger than 1 MByte.

A potential checkpoint location is placed within the outer iteration loop. It is thus visited $1,000$ times.

T_c $[s]$	# of ckpts	UCF compatible			UCF incompatible		
		t_{chkpt}	ovh	t_{rec}	t_{chkpt}	ovh	t_{rec}
∞	0	196.1	7	196.1	196.1	7	196.1
128	1	198.0	8	201.3	198.0	8	202.6
64	3	200.4	9	207.2	201.9	10	209.5
32	6	205.2	12	215.0	207.8	13	220.5
16	12	213.0	16	231.6	218.7	19	240.7
8	24	228.9	25	262.3	240.5	31	282.7
4	47	262.9	44	331.6	285.2	56	374.1
2	91	324.2	77	453.2	363.0	98	539.0
1	168	430.3	135	684.4	505.3	176	846.5
0	1000	1662.5	807	—	1996.5	990	—
runtime without instrumentation: 183.2 s							

(a) IBM Thinkpad i486DX475 / Linux

T_c $[s]$	# of ckpts	UCF compatible			UCF incompatible		
		t_{chkpt}	ovh	t_{rec}	t_{chkpt}	ovh	t_{rec}
∞	0	62.1	0	62.1	62.1	0	62.1
32	1	62.3	1	62.7	62.5	1	62.8
16	3	62.6	1	63.7	62.7	1	64.1
8	7	62.9	2	65.8	63.4	3	66.7
4	15	63.8	3	69.9	64.9	5	71.9
2	30	65.6	6	77.6	67.5	9	81.5
1	59	68.8	11	92.5	72.7	18	100.2
0	1000	173.7	181	—	241.0	290	—
runtime without instrumentation: 61.8 s							

(b) Sun SPARCstation20 / SunOS5.3 (Solaris)

Figure 9: Heat equation on two systems, storing UCF checkpoints on the local disk. T_c is the checkpoint timer interval. Runtimes t_{chkpt} without failures and t_{rec} in the presence of failures are given in seconds, the overhead of checkpointing (ovh) given in per cent (t_{chkpt} with respect to the runtime without instrumentation).

Figure 9 summarizes the results of our experiments on an IBM Thinkpad 701 based on an Intel 486DX475 processor operated by Linux, and on a Sun SPARCstation20 running SunOS 5.3, with checkpointing to local disk. We measured the runtimes for a range of timer intervals leading to different numbers of checkpoints during execution, for UCF compatible and UCF incompatible checkpointing, and including failures and recovery.[3] Measurements are denoted UCF compatible, if the UCF specification matches the system architecture. For UCF incompatible checkpointing, alignments and conversions, involving swapping the byte sex, are performed on the i486 to match the format of the SPARCstation and vice versa.

Figure 9 illustrates how often checkpoints can be saved without affecting performance substantially. The overhead is less than 10 %, if the checkpointing interval T_c larger than 32 seconds on the Thinkpad, and larger than 1 second on the SPARCstation. Although these values depend on the checkpoint size, they are small compared to typical system MTBF values. Note that the conversion penalties paid for UCF incompatibility are only severe if the checkpointing frequency is unrealistically high.

The columns labeled t_{rec} in Figure 9 give the minimum run times of the program, if one failure occurs per checkpointing interval. This "ideal" failure situation is simulated by exiting the program just after a checkpoint has been stored, capturing the exit status within a shell script that immediately invokes the program again with the recover option enabled. Since the program is aborted immediately after a checkpoint is stored, no replay of lost computation is required. Furthermore, the time for failure detection as well as downtimes are (almost) zero. Since the state is recovered from local disk, no overhead is incurred by transferring the checkpoint via the network.

A single recovery on a UCF compatible architecture costs about 2 s on the i486, and about 0.4 s on the SPARCstation20. These numbers are dominated by the use of the local disk as stable storage for the checkpoint. Both systems suffer from an overhead penalty due to data representation conversion during recovery. The difference between the runtimes of the recovered experiments with UCF incompatible architectures and UCF compatible architectures gives the overhead of two conversions, one during checkpointing and the other during recovery.

The conclusion of these experiments is that the checkpointing overhead is negligible for reasonable checkpointing frequencies, even when conversion into UCF representation is required.

6 Limitations

The generation of code to save and recover portable checkpoints by means of source-to-source compilation is a powerful and versatile method. However, the approach has its limitations. We first identify problems that will limit any solution for the problem of portable checkpointing of C programs:

- Use of non-portable features in programs: If checkpoints are to be portable, it is essential that

[3] Note that code instrumentation introduces a runtime penalty on the i486 for $T_c = \infty$, but does not affect the runtime on the SPARCstation.

the programs being checkpointed themselves be portable.

- Loss in floating point accuracy due to data representation conversion: This problem can only be addressed by conformance to standards.
- Ambiguous type information when generating checkpointing code: If variables, for example, are declared as integers and casted to pointers, the checkpoint is likely to be incorrect. A similar ambiguity arises when interpreting the value of a `union` via fields of different type. This problem would not arise in programming languages with a strict type system.
- Functions with side effects: If a function in a call sequence to a checkpoint causes side effects, and is called in expressions such as `if` conditions, it may not be possible to instrument such function calls for checkpointing without changing the program semantics. We expect the programmer to clean up the code, if `c2ftc` detects such a situation.

The following represent `c2ftc` specific limitations:

- We did not address file I/O and interprocess communication for `c2ftc` yet. We expect to provide portability based on the approach of logging *message determinants* [1], which is applicable to both file I/O and interprocess communication.
- Our current runtime support is targeted at Unix dialects. There is no restriction, in principle, in adapting it to other systems, although a performance penalty may be incurred at runtime.
- The described approach assumes that the application program uses the memory allocation functions supplied by `c2ftc`, which transparently replace the C memory management library routines. If the application accesses allocated memory incompatible with the type information supplied in the allocation request, the runtime library will be unable to perform the saving, restoring and conversion of heap data correctly.

7 Conclusion

We have introduced the concept of portable checkpoints, and presented a source-to-source compiler approach to implement portable checkpoints for heterogeneous computer networks. Furthermore, we have demonstrated that the overhead introduced by portable checkpointing is very low when reasonable checkpoint intervals are chosen.

The proposed compiler approach only requires that (1) a user program be submitted to a front-end source-to-source C compiler before compilation on the desired target machine, and (2) a run time library be linked to produce the final executable. It does not limit the choice of compiler or impose any system-specific demands. This makes it easy to render a large subset of C programs robust in the presence of faults and recoverable on any UNIX-based system.

References

[1] Alvisi L., Hoppe B., Marzullo K. Nonblocking and Orphan-Free Message Logging Protocols. In *23rd Fault Tolerant Computing Symposium*, pages 145–154, Toulouse, France, June 1993.

[2] Elnozahy E.N., Johnson D. B., Zwaenepoel W. The performance of consistent checkpointing. In *IEEE Symposium on Reliable and Distributed Systems*, pages 39–47, October 1992.

[3] Franz M. *Code Generation on the Fly: A Key to Portable Softare*. PhD thesis, Institute for Computer Systems, ETH Zurich, 1994.

[4] Gosling J. The Java Language Environment. Technical report, Sun Microsystems, Mountain View, California, 1995. White Paper.

[5] Li C-C. J., Stewart E.M., Fuchs W.K. Compiler Assisted Full Checkpointing. *Software - Practice and Experience*, 24 no. 10:871–8861, October 1994.

[6] Plank J.S., Beck M., Kingsley G., Li K. **Libckpt:** Transparent Checkpointing under Unix. In *Proceedings of the Usenix Winter Technical Conference*, San Francisco, CA, January 1995.

[7] Seligman E., Beguelin A. High-Level Fault Tolerance in Distributed Programs. Technical Report CMU-CS-94-223, Carnegie-Mellon University, December 1994.

[8] Strumpen V. Software-Based Communication Latency Hiding for Commodity Networks. In *International Conference on Parallel Processing*, August 1996.

[9] Strumpen V., Ramkumar B. Portable Checkpointing and Recovery in Heterogeneous Environments. Technical Report 96-6-1, Dept. of Electrical and Computer Engineering, University of Iowa, June 1996.

[10] Theimer M.M., Hayes B. Heterogeneous Process Migration by Recompilation. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 18–25, July 1991.

[11] Zhou S., Stumm M., Li K., Wortman D. Heterogeneous Distributed Shared Memory. *IEEE Transactions on Parallel and Distributed Systems*, 3 no. 5:540–554, September 1992.