Portable Checkpointing and Recovery in Heterogeneous Environments

Volker Strumpen and Balkrishna Ramkumar
Department of Electrical and Computer Engineering,
University of Iowa, Iowa City, Iowa 52242
email: {strumpen,ramkumar}@eng.uiowa.edu

Abstract

Current approaches for checkpointing and recovery 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 the failed computation on a different processor architecture, with possibly different byte-ordering and data-alignment specifications. This implies that checkpointing and recovery must be *portable*. We provide portability by means of a universal checkpoint format that allows object codes to resume execution from a checkpointed state, allowing for fast execution of already compiled code, rather than interpreting or compiling on the fly. This paper describes the system support needed to implement *portable checkpoints*, and the *shadow checkpoint algorithm* to checkpoint and recover a sequential process. Experimental results on three different architecture-operating system combinations demonstrate the checkpointing overhead and the cost of recovery.

1 Introduction

As Internetworking matures, worldwide distributed computing will soon enter the realm of possibility. Simple probabilistic analysis suggests that such large geographically distributed systems will exhibit a high probability of single failures, even if each individual component is quite reliable. Due to the difficulties associated with programming such systems today, local area networks (LANs) are still used heavily for long running simulations. Even on such systems, failures occur frequently [23] due to a variety of reasons including network failures, processor failures, and even administration downtimes. Thus fault tolerance is fast becoming an essential feature for networked programming systems.

Large distributed systems are inherently heterogeneous in nature. Even LANs today often consist of a mixture of binary incompatible hardware components and operate an even larger variety of operating systems. Providing 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.

In this paper, we present a new efficient checkpointing and recovery mechanism that provides both portable program execution as well as fault tolerance in heterogeneous environments. Other approaches to portable fault tolerant computing have been investigated. Languages like Java [8] 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 magniture slower than comparable C programs. Another possibility is "compilation on the fly" [7] 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, neither interpreter-based systems nor compilation on the fly are explicitly designed to support fault tolerance. Providing hardware support for fault tolerance such as Sheaved Memory [21] or the Virtual Checkpoint Architecture [3] are not designed for portability either.

The remainder of this paper is organized as follows. In Section 2, we describe related work in the area of checkpointing and recovery. We then outline our objectives in Section 3, thereby providing a basis for the rest of the discussion in this paper. Section 4 describes the system support for portable checkpoints in our scheme. Our algorithm is described in detail in Section 5. In Section 6 we present experimental evidence that the loss of efficiency traded in for portability is acceptable.

2 Related Work

Much of recent related work is in the area of checkpointing distributed programs, for example [2, 4, 5, 6, 9, 11, 16, 19, 20]. In addition, projects described in [10, 12, 15, 18] concentrate on shared or distributed shared memory (DSM) systems. Many of the optimizations discussed in the work cited above are applicable to checkpointing sequential programs. We restrict our discussion below to these optimizations.

Previous work has focussed on the runtime and space efficiency objectives. Li and Fuchs [12] were among the first to demonstrate the use of compilers to identify potential checkpoints in programs. At runtime, heuristics are used to determine which of these checkpoints will be activated. Plank and others [13] propose latency hiding optimizations to reduce the cost of transformation of a checkpoint to stable storage.

Elnozahy [6] and Plank [17] have proposed efficient implementation techniques that limit the overhead of checkpointing to within 2-5% of the the overall runtime of a program, when checkpoints are taken every two minutes. These techniques rely on efficient page-based bulk copying and hardware support to identify memory pages since the last checkpoint. However, these optimizations are restricted to binary compatible hardware and operating systems. Thus, their applicability in a heterogeneous system is severely limited.

Seligman and Beguelin [19] have developed checkpointing and restart algorithms in the context of the Dome C++ environment. Dome provides checkpointing at multiple levels, ranging from high level user-directed checkpointing that sacrifices transparency for portability and low overhead, to low level checkpointing that is transparent but results in non-portable code and requires larger checkpoints. Dome's high-level checkpointing is designed for portability, but restricts the programming model to achieve this goal.

3 Objectives and Hazards

We consider the following properties as primary objectives, listed by decreasing priority:

Portability: The primary requirement is a portable checkpoint. As machines crash, it is often unlikely that another, binary compatible processor is available to continue the job. Even in distributed systems, it is generally prefered to restart a job on a machine at a close proximity to the crashed machine to exploit various kinds of locality.

Runtime overhead: Both checkpointing and recovery should not increase the runtime of the program significantly. For checkpointing at reasonable time intervals, the overhead on overall execution time should be kept below, for example, 10%, depending on the volume of data to be checkpointed.

Space overhead: The size of checkpoints should be minimized. In a real system, the available memory or disk space may limit the amount of checkpointing, and, in turn, even the problem size of the program.

Transparency: Ideally, the programmer should need to do no more than use a precompiler[1] and link a runtime library with the application object code in order to render the executable fault tolerant. A precompiler should be able to analyze and insert checkpoint requests at suitable points in the user code.

We concentrate on C programs in this paper. We believe that the flexibility afforded by pointers in C provides the real technical challenge when supporting portable checkpointing. We assume that any program under consideration has been written in a portable manner. In particular, the use of system calls, library functions, and so on, has been handled with care. Our *portable checkpoints* are designed to cope with the following portability hazards:

Architecture: The primary obstacle for portability is the variety of representations of basic data types and the data layout in memory.

Data representation: Although most recent architectures support the IEEE floating point standard 754, there is no agreement about implementing little endian or big endian memory addressing. Furthermore, the number of bits invested into the representation of basic data types varies, and is currently changing with the introduction of 64-bit architectures.

Alignment: Different memory architectures require different alignments. Since compiler writers tend to optimize space efficiency, alignment computations optimize the underlying memory architecture to a certain extend. As a consequence, data objects can have different sizes on different architectures, even if the data representations are identical.

Hardware Support for Programming Languages: The number of available registers and the organization of the register file significantly affect the structure of the runtime stack. For example, the implementation of *register windows* leads to a unique stack frame layout on SPARC architectures. Moreover, different compilers exploit architectural features differently. As a result, the runtime stack layout varies from system to system.

UNIX implementation: Differences among UNIX implementations (BSD/System V) complicate checkpoint portability.

[1] Our precompiler is a source-to-source compiler that generates C code, which can be compiled by any native C compiler.

Address space: The address space, or memory layout, of a UNIX process is not portable. Besides the fact that different page sizes are used, the quasi-standard address space places the text and data/bss segments into the bottom, grows the heap upwards above these segments, and the stack downwards, its bottom aligned to the upper end of the space. The HPUX operating system, on the other hand, divides the address space into four quadrants, places the text segment into the first quadrant, data/bss and heap into the second, and grows the stack upwards starting near the middle of the second quadrant. Page-based checkpointing cannot provide portability across different address space layouts.

System calls: Different UNIX systems may provide access to the same functionality by means of different names (getpagesize/sysconf), or provide system calls with the same name but different functionalities (mprotect). Whereas some operating systems allow for protecting any page of a process's virtual address space at user-level, some only provide this feature for memory-mapped areas. The later ones do not facilitate the implementation of page-based checkpointing schemes.

Language features: Certain programming language features yield runtime behavior that is not portable.

Pointers into a process's address space are in general not portable when considering different architectures or different address space layouts.

Dynamic memory management is system specific, due to differences in allocation schemes.

4 Portability Structures

The design trade-offs in checkpoint and recovery schemes involve both system and program properties. In particular, the mean time between failure (MTBF) can be viewed as a system property, assuming that the failure is not caused by the application program. On the other hand checkpoint size is primarily dependent on the program, and the points in the program execution where the checkpoint is taken. Therefore, we distinguish *potential checkpoint locations* of a program from checkpoints that are actually stored. Potential checkpoint locations are program specific and may be chosen by a compiler or user in order to minimize checkpoint size. The optimal frequency of checkpointing depends on the underlying system's MTBF. Assuming that potential checkpoint locations are reached relatively often during MTBF, the minimum time between checkpoints (MinTBC) can be optimized based on the system's MTBF. Thus, a checkpointing facility should provide (1) a construct to label potential checkpoint locations, for example, by means of a function call, and (2) a timer-based decision control about when a potential checkpoint location should be activated if visited at runtime. With timer-based activation, MinTBC equals the timer interval. After the timer expires, a checkpoint will be stored at the first potential checkpoint location encountered.

We introduce the following concepts to support portable checkpoints:

1. Shadow address space is a user-level partitioning of a process's virtual address space. Its functions are to (1) reserve memory for the shadow stack, which can be viewed as a marshaling buffer for the checkpoint, (2) support user-level heap allocation to simplify pointer resolution, and (3) provide performance optimization by using interrupts to handle shadow stack overflows.

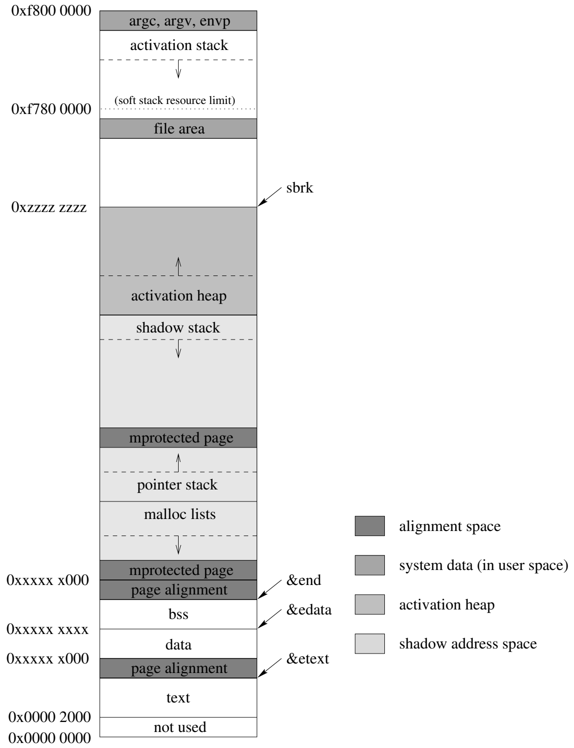


Figure 1: Shadow address space.

Figure 1 shows the organization of a typical shadow address space. At the bottom are the text, data and bss segments, whereas the stack is aligned to the top of the address space. We use the heap area to allocate a sufficient amount of memory that holds the *activation heap*, the *shadow stack* and the data structures needed for managing memory and checkpointing. These include the *pointer stack*, explained below, and the *malloc lists* of our user-level memory management of the activation heap. To distinguish the runtime stack from the shadow stack, it is called *activation stack* in the following. The shadow stack is the central data structure for checkpointing. All variables defining the state of the program at the time of checkpointing are pushed onto the shadow stack. Eventually, the shadow stack holds the checkpoint in a contiguous area of memory, that can be transferred to stable storage.

The pointer stack and malloc lists are dynamic data structures. We use write protected virtual memory pages to handle overflows efficiently. On systems that do not allow for write protecting virtual memory pages at user-level, overflow checks need to be included in the shadow stack and memory allocation operations.

Malloc lists are managed by a heap memory management scheme that maintains more information about a heap object than the operating system is necessary for checkpointing and recovering pointers. During program translation, functions are generated to explicitly save, convert and restore data in each stack frame, each dynamically allocated data type, and global data in the program. References to these functions are associated with each allocated memory block.

An important reason for introducing the shadow stack has been the fact that transferring a large checkpoint via a network is the most time consuming portion of checkpointing. Once the checkpoint is available on the shadow stack, communication latency hiding can be used to transfer the checkpoint and continue the computation at the same time [22].

2. Universal Checkpoint Format (UCF) specifies the layout of a portable checkpoint, such as header information and data segments, as well as the data representations and alignments used in the checkpoint. UCF data representations and alignments can be specified by the user. As such, UCF is an adaptable format that can be customized to particular networks. For example, data representations and alignments of the majority of available machines in the network can be chosen. Since UCF-incompatible systems have to convert data types, UCF-incompatible systems pay a runtime penalty. The concept of UCF for portable checkpoints is similar to, but more flexible than, the external data representation (XDR) for remote procedure calls [14].

```
typedef struct {            typedef struct {
  char   c;                   char   c;
  double d;                   int    pad;
} cd_t;                       double d;
                            } cd_t;
```

Figure 2: UCF padding of source code.

Alignment incompatabilities pose special problems on portability. Consider the example in Figure 2. The size of structure `cd_t` on the left, depends on the data representations and alignment. For example, on both a i486/Linux system and a Sparc/Sunos system, a `char` consumes 1 byte and a `double` 8 bytes. However, on the former system, a `double` is 4 byte aligned leading to a structure size of 12 bytes, whereas a `double` on the later system is 8 byte aligned, yielding a structure size of 16 bytes. To cope with such alignment incompatibilities, the compiler generates padding variables into the source code according to the UCF alignment. The declaration in Figure 2 on the right introduces the integer `pad` for this purpose. The benefits of uniform structure sizes during checkpointing and recovery outweigh the possible loss of space efficiency incurred on all UCF-incompatible systems. All UCF data representations and alignments must be at least as large as the maximum values on all systems potentially being used. Data representation conversion, which happens on all systems incompatible to UCF, is done in-place on the shadow stack.

5 Shadow Checkpointing

During checkpointing, all program data, including global data, heap data and stack data are systematically saved contiguously on the shadow stack. The handling of stack variables is one of the most interesting aspects of our algorithm and is described in detail below. Global and heap variables are treated later.

We begin by first considering only non-pointer variables on the activation stack. The algorithm is then extended in Section 5.2 to support pointer variables. We illustrate the checkpointing and recovery scheme by means of a recursive program to compute Fibonacci numbers shown in Figure 4. The 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 `main` function by renaming, and function `fib`. We assume that a potential checkpoint location is specified within `fib` by means of a call to function `checkpoint`.

Reduction of checkpoint size and thus overall checkpointing overhead is based on the identification of the state of a program at a potential checkpoint location at compile time. The state consists of the set of existential variables at a given potential checkpoint location:

Definition (Existential Variable) *A variable is existential at a potential checkpoint location l_c, if it is assigned before l_c, and appears in an expression after l_c.*

The basic idea for saving the existential variables of the stack is to visit all stack frames, and save the existential

```
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 3: Code fragment illustrating the call sequence of the Shadow Checkpoint Algorithm; cf. Figures 4 and 5.

variables specified at compile time. For a portable implementation, only function call and return instructions can be used safely. Consequently, checkpointing the stack is implemented by returning the function call sequence, thereby visiting each individual stack frame, and restoring the stack by calling the same function call sequence again. This scheme allows for identifying existential variables at compile time, accessing each variable individually rather than block-copying the stack, and avoids non-portable implementations based on setjmp/longjmp pairs, as for example `libckpt` [17].

In order to preserve the program's state while checkpointing, none of the program's statements may be executed. Therefore, function calls and returns for checkpointing require a modification of the original program. Source-to-source compilation transforms the code fragment given in Figure 3 into the code shown in Figure 4. This transformation is derived such that checkpointing does not interfere with normal program execution.

The code transformation is based on the distinction of the following three phases of checkpointing:

Save phase: The existential data of all user stack frames[2] are pushed onto the shadow stack. Note, that the set of existential variables of a function can differ, if, for example, `checkpoint` is called more than once in the same function in different locations.

Restore phase: Since, in general, a stack frame's contents are lost after a function return, the stack needs to be restored. Therefore, the original stack is rebuilt by executing the reverse sequence of function calls of the return sequence during the save phase. Preserving the program's state at the time `checkpoint` was called, no state changing statements are executed. Only existential data are copied from the shadow stack back into the activation stack frame.

Store phase: Once the state is restored from the shadow stack, the contents of the shadow stack constitutes a checkpoint, which can be written to stable storage or remote memory. This store can be done simultaneously with subsequent program execution.

The implementation of the store phase involves issues beyond the scope of this paper. Instead, we focus on the save and restore phases. We now present the *shadow checkpoint algorithm* for checkpointing. Figure 5 illustrates the execution of the algorithm based on the code fragments in Figures 3 and 4. The lower part of Figure 4 shows the macros and functions used to save, restore, and convert the checkpoint of the fibonacci example when run on an i486 architecture under Linux. The UCF data representation of integers and unsigned long is big endian, and uses 4 bytes. Since the i486 is a little endian architecture, these data types have to be byte swapped (see macro `_SL_conv_word`). Actually, for the runtimes reported in Section 6, we used the `bswap` instruction of the i486 instead of the given macro.

Algorithm 1 (Shadow Checkpoint) *The shadow checkpoint algorithm saves the existential variables of a process into the shadow stack using only function calls and returns to change the control flow. Variable* `chkptmode` *holds the state of the checkpointing algorithm.*

1. *If* `checkpoint` *is called, checkpointing is timer activated, and* `chkptmode=EXECUTE`*, i.e. normal program execution, push the data/bss and heap segments onto the shadow stack, set* `chkptmode=SAVE`*, and return to the caller.*

2. *If, after a function returns to its caller other than* `main`*, and* `chkptmode=SAVE`*, push the existential portion of the local (caller's) stack frame, including the function identifier* `funid` *of the returner (cf. Figure 4), onto the shadow stack and return. Step 2 is repeated until control returns to main.*

3. *If control returns to* `main` *and* `chkptmode=SAVE`*, set* `chkptmode=RESTORE` *and call function* `chkpt_main` *again.*

4. *If a function other than* `checkpoint` *is called, and* `chkptmode=RESTORE`*, pop the existential portion of the local stack frame from the shadow stack and call the returner. The function identifier, saved in step 2, is used as key for a jump table lookup (see*

[2] User stack frames are stored between, and excluding the frames of `main` and `checkpoint`.

```
#include <stdio.h>
#include "chkpt.h"
#include "fib_chkpt.h"

extern int    checkpoint();

chkpt_main(int argc, char *argv[])
{
   unsigned long _SL_funid, _SL_addr;
   int n, result;

   switch (_SL_chkptmode) {
     case(_SL_EXEC): break;
     case(_SL_RESTORE):
       _SL_addr = s_stack.top;
       _SL_RESTORE_chkpt_main_0;
       _SL_CONVERT_chkpt_main_0(_SL_addr);
       switch(_SL_funid) {
         case(0): goto L_SL_fun0;
       }
     case(_SL_RECOVER):
       _SL_addr = s_stack.top;
       _SL_CONVERT_chkpt_main_0(_SL_addr);
       _SL_RESTORE_chkpt_main_0;
       switch(_SL_funid) {
         case(0): goto L_SL_fun0;
       }
   }

   n = atoi(argv[1]);

   _SL_funid = 0;
L_SL_fun0:
   result = fib(n);
   switch (_SL_chkptmode) {
     case(_SL_EXEC):   break;
     case(_SL_SAVE):
       _SL_SAVE_chkpt_main_0;
       return 0;
   }
   printf("\n fib(%d) = %d\n", n, result);
}

int fib(int n)
{
   unsigned long _SL_funid, _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_funid) {
         case(0): goto L_SL_fun0;
         case(1): goto L_SL_fun1;
         case(2): goto L_SL_fun2;
       }
     case(_SL_RECOVER):
       _SL_addr = s_stack.top;
       _SL_CONVERT_fib_0(_SL_addr);
       _SL_RESTORE_fib_0;
       switch(_SL_funid) {
         case(0): goto L_SL_fun0;
         case(1): goto L_SL_fun1;
         case(2): goto L_SL_fun2;
       }
   }

   if (n > 2) {
     _SL_funid = 0;
L_SL_fun0:
     _SL_fun0  = fib(n-1);
     switch (_SL_chkptmode) {
       case(_SL_EXEC):   break;
       case(_SL_SAVE): _SL_SAVE_fib_0;
               return 0;
     }

     _SL_funid = 1;
L_SL_fun1:
     _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_funid = 2;
L_SL_fun2:
     checkpoint();
     switch (_SL_chkptmode) {
       case(_SL_EXEC):   break;
       case(_SL_SAVE): _SL_SAVE_fib_0;
                         return 0;
     }
     return 1;
   }
}
```

```
#define _SL_conv_word(addr) { \
   _SL_tc                  = *((char *) (addr)); \
   *((char *) (addr))      = *(((char *) (addr))+3); \
   *(((char *) (addr))+3)  = _SL_tc; \
   _SL_tc                  = *(((char *) (addr))+1); \
   *(((char *) (addr))+1)  = *(((char *) (addr))+2); \
   *(((char *) (addr))+2)  = _SL_tc; \
}

#define _SL_SAVE_chkpt_main_0 { \
  *--((unsigned long *) s_stack.top) = _SL_funid; \
  *--((int *) s_stack.top) = n; \
}

#define _SL_RESTORE_chkpt_main_0 { \
  n = *((int *) s_stack.top)++; \
  _SL_funid = *((unsigned long *) s_stack.top)++; \
}

static __inline__ void _SL_CONVERT_chkpt_main_0(addr)
unsigned long addr;
{
  _SL_conv_word(addr);   ((int *) addr)++;
  _SL_conv_word(addr);   ((unsigned long *) addr)++;
}

#define _SL_SAVE_fib_0 { \
  *--((unsigned long *) s_stack.top) = _SL_funid; \
  *--((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_funid = *((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-instrumented code fragment derived from Figure 3. The include file `fib_chkpt.h` is given in the lower part.

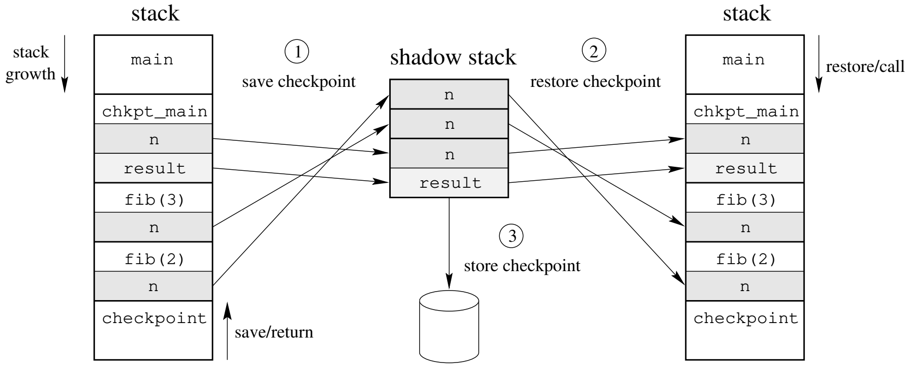


Figure 5: Shadow checkpointing the stack. The three phases of the algorithm *save*, *restore* and *store* are indicated.

`RESTORE` *case of the function entry switches in Figure 4), to jump to the returners function call.*

5. *If* `checkpoint` *is called and* `chkptmode=RESTORE`*, the checkpoint is stored. Then,* `chkptmode` *is reset to* `EXECUTE` *and function* `checkpoint` *returns to program execution.*

The save phase begins when function `checkpoint` is called. All functions on the stack save their frames and return until `main` becomes active. Therefore, steps 1–3 belong into the save phase. `main` also starts the restore phase, which is finished when function `checkpoint` is called again. Thus, steps 3–5 belong to the restore phase.

All variables are accessed by name to save and restore their values to and from the shadow stack. To be in lexical scope `_SL_SAVE_<fun>`, `_SL_RESTORE_<fun>`, and `_SL_CONVERT_<fun>` must be defined as macros, which also has a performance benefit over function calls. A naming scheme is necessary to distinguish the macros called in different functions and used for multiple function calls of the same functions within a single function. Therefore, `fun` is replaced by a string, generated by concatenating a function name and the identifier of the potential checkpoint location.

5.1 Shadow Recovery

Recovery from a checkpoint comprises essentially the restore phase of the Shadow Checkpointing Algorithm. Before the restore phase, the checkpoint must be block-copied from secondary storage into the shadow stack. Then, the computation is restored from the shadow stack:

Load phase: After restarting the executable, load the checkpoint into the shadow stack.

Restore phase: First, the data/bss and heap segments are restored. Then, the stack is rebuilt by executing the reverse sequence of function calls of the return sequence active during checkpointing. This sequence is encoded in the local variables called `_SL_funid`, which save the information of the next function to be called in the checkpoint.

The following algorithm is used to recover a computation from the shadow stack:

Algorithm 2 (Shadow Recovery) *The shadow recovery algorithm restores the existential variables of a process from the shadow stack using only function calls and returns to change the control flow.*

1. *If the program is started and the recover option specified, read the checkpoint into the shadow stack, and restore the data/bss and heap segments.*

2. *In* `main`, `chkptmode` *is set to* `RECOVER`*, and* `chkpt_main` *is called.*

3. *If a function other than* `checkpoint` *is called, and* `chkptmode=RECOVER`*, pop the existential portion of the local stack frame from the shadow stack and call the returner. (Same as step 4 of Algorithm 1.)*

4. *If* `checkpoint` *is called and* `chkptmode =RECOVER`, `chkptmode` *is set to* `EXECUTE` *and* `checkpoint` *returns.*

Steps 2–4 are almost the same than the restore phase of shadow checkpointing. The only difference is that data representations are converted on the shadow stack before restoration. In Figure 4, the sequence of the `_SL_CONVERT_<fun>` and `_SL_RESTORE_<fun>` macros is reversed in the `RESTORE` and `RECOVER` cases.

5.2 Pointers

We classify pointers by means of 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 their addresses and size differ from target to target:

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.[3]

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 does 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.

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.

The second category classifies pointers 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.

Furthermore, there are two kinds of stack pointers:

1. Pointers to objects within the scope of the function the pointers are declared in. The pointer and the target object are allocated within the same stack frame. We call these pointers *intra-frame pointers*.

2. Call-by-reference parameters are pointers into an anchestor frame on the activation stack. We call these pointers *inter-frame pointers*.

During execution, the stack frame (the callee frame) containing a pointer passed as a parameter is always pushed onto the activation stack above 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.

[3] We could also choose to offset against the bottom of the stack image on the shadow stack. However, using the displacement simplifies the distinction of forward and backward pointers during recovery.

5.3 Stack Pointers

During checkpointing, it is necessary to identify the location of the target of a pointer on the shadow stack. The basic idea is to save the pointer as portable offset within the UCF-organized shadow stack. We first consider pointer variables on the stack that point elsewhere on the stack.

5.3.1 Checkpointing Stack-Located Stack Pointers

The following algorithm extends Algorithm 1 with the handling of stack pointers.

Algorithm 3 (Stack Pointers) *This algorithm incorporates the handling of stack pointers into Algorithm 1.*

1. *Same as Shadow Checkpoint Algorithm.*

2. *Pointers are not only saved on the shadow stack, but additionally the shadow stack address of each pointer is pushed onto the pointer stack.*

3. *Same as Shadow Checkpoint Algorithm.*

4. *When restoring an object that is potentially pointed to, additionally (1) check whether an address on the pointer stack lies within the address range of this object on the stack. If yes, while restoring the object (2) save the base address of the object's copy on the stack in the shadow stack by overwriting (parts of) the object itself. (3) Calculate the portable shadow stack displacement between the pointer's shadow stack address and the pointer's target shadow stack address, and store it on the pointer stack.*

5. *In* `checkpoint`, *for each shadow address on the pointer stack, overwrite the contents of the shadow address by its corresponding portable offset.*

As an example, consider the code fragment in Figure 6 and the illustration in Figure 7.

```
extern int checkpoint();

chkpt_main()
{
   long a[4];

   function1(a);
}

function1(long *p)
{
   p += 2;
   *p = 1;
   checkpoint();
}
```

Figure 6: Code fragment illustrating the Shadow Checkpoint Algorithm with call by reference; cf. Figure 7.

When `checkpoint` is called, the save phase begins. First, the existential variables of `function1` are pushed onto the shadow stack, in particular pointer `p`. According to Figure 7, `p` is stored on the stack at X_p, and pushed into X_{ps} on the shadow stack. Additionally, 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. The address of the array element `p` is pointing to is marked X, and its shadow X_s. No special action is necessary when copying the pointer target object to the shadow stack. Note that objects can be copied as entities, because they are UCF aligned. Consequently, array `a` is not pushed element-wise, which would reverse the order of the elements on the shadow stack.

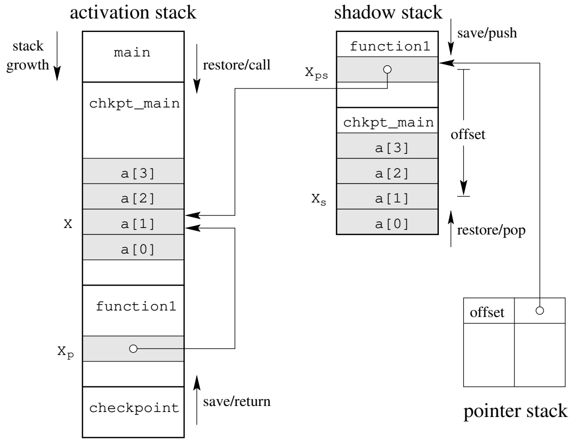


Figure 7: Checkpointing the stack in the presence of a call-by-reference.

After saving, the restore phase begins. The stack is restored by calling the returned functions and popping the elements of the stack frames from the shadow stack. First, the frame of `checkpoint` is restored. Before restoring array `a`, the pointer stack is checked for a reference into `a` on the stack. In this example, the pointer in X_{ps} points to address X. Now, the shadow offset can be computed according to the rule

$$\text{offset} = \text{pointer target address} - \text{pointer address}, \quad (1)$$

where both addresses are shadow stack addresses. In Figure 7, offset $= X_s - X_{ps}$. X_{ps} is given on the pointer stack. Determining X_s requires some offset computation: The offset of X to the base of the object $\&a[0]$ is known, because X is accessed indirectly via the pointer stack, and a is known by name. The shadow stack pointer currently points to the copy of the base element of a on the shadow stack. Thus, $X_s = ssp + X - a$, where *ssp* stands for the shadow stack pointer. The offset is stored on the pointer stack associated with the pointer to X_{ps}. We cannot overwrite X_{ps} immediately, because it holds the value of pointer `p`, which is needed, when restoring the stack frame of `function1`. Only after the entire stack is restored, a sweep through the pointer stack copies the offsets into the addresses on the shadow stack. Offset $X_{ps} - X_s$ will overwrite the value of `p` in address X_{ps}.

5.3.2 Recovery of Stack-Located Stack Pointers

Although conceptually recovery from a checkpoint is 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 8 illustrates the recovery from the checkpoint in Figure 7.

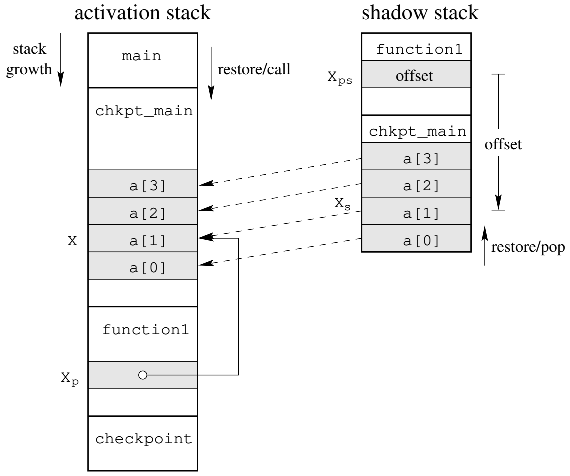


Figure 8: Recovery of the stack in the presence of a call-by-reference.

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 activation stack is stored in its location on the shadow stack after the value of the element has been restored on the activation stack; cf. broken lines in Figure 8. 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`$= \left[X_{ps} + [X_{ps}]\right]$. The pointer can be found in the shadow stack address which is computed according to the rule:

$$\text{pointer address} = \text{shadow pointer address} + \text{offset}. \quad (2)$$

This simple lookup is bought by saving the complete mapping of the restore target addresses on the activation stack in the shadow stack. This expense is justified by the fact, that recovery will be the infrequent case, and that another scheme would require a substantially more complex treatment of pointers.

5.3.3 Data/Bss and Heap-Located Stack Pointers

So far, we treated stack pointers located in the stack. We now consider stack pointers located in data/bss and the heap. Figure 9 shows a stack pointer located in data/bss ($X_{dbp} \rightarrow X_{sdb}$) and a stack pointer located on the heap ($X_{hp} \rightarrow X_{sh}$). Since the order chosen to push segments onto the shadow stack is data/bss before heap before stack (cf. Algorithm 1), both stack pointers are forward pointers. Forward pointers are resolved by means of a pointer stack analogous to the handling of stack pointers located in the stack as described above.

The following actions are added to the save and restore phases of the stack to handle the stack pointers in Figure 9 located in data/bss and the heap:

Save phase: Additionally to saving the existential variables on the shadow stack, push pointers to the shadow copies of X_{dbp} and X_{hp} in X_{dbps} and X_{hps} onto the pointer stack.

Restore phase: When restoring stack pointer targets (X_{sdb} and X_{sh}), calculate the corresponding offsets — the shadow stack pointer points to the target, used to recognize that the offset needs to be computed, and the pointer stack pointer to the pointer's shadow copy itself — and save them on the pointer stack. After the shadow stack has been restored, substitute the offsets for all pointers. In Figure 9, assign $X_{dbps} \leftarrow \text{offset}_{db} = X_{sdbs} - X_{dbps}$ and $X_{hps} \leftarrow \text{offset}_h = X_{shs} - X_{hps}$.

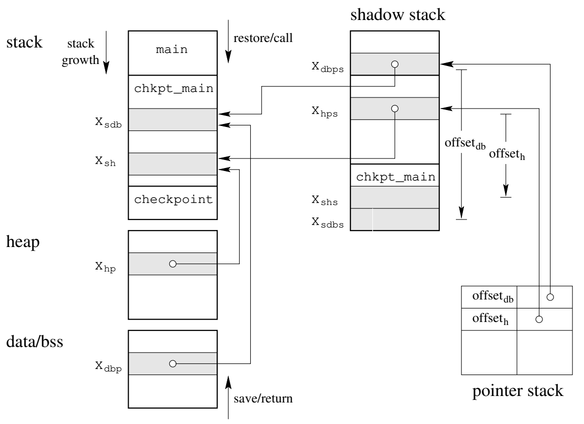


Figure 9: Checkpointing in the presence of *forward stack pointers* from heap and data/bss segments into the stack.

Recovery from the checkpoint containing the stack pointers of Figure 9 is illustrated in Figure 10.

Restore phase: First, data/bss and heap segments are restored. When arriving at a pointer, i.e. the offset on the shadow stack, a pointer to the corresponding target as well as to the pointer's origin in the activation space, accessed by name, is pushed onto the pointer stack. For example, when arriving at address X_{dbps} in Figure 10, pointers to the origin of $X_{dbps} = X_{dbp}$ and $X_{sdbs} = X_{dbps} - [X_{dbps}]$, where $[X_{dbps}] = \text{offset}_{db}$ are pushed onto the pointer stack. X_{hps} is handled analogous.

After an object is restored, the pointer stack is checked to find those pointers pointing to the object. Here, the pointers in X_{dbps} and X_{hps} are forward pointers, and can be resolved in a single phase, because the targets are visited after the pointers themselves. When restoring the shadow addresses X_{shs} and X_{sdbs}, the pointers on the pointer stack indicate that these are pointer targets. Since the corresponding addresses on the activation stack can be accessed by name, the pointers in X_{hp} and X_{dbp}, accessible from the pointer stack, are assigned these addresses (X_{sh}, X_{sdb}).

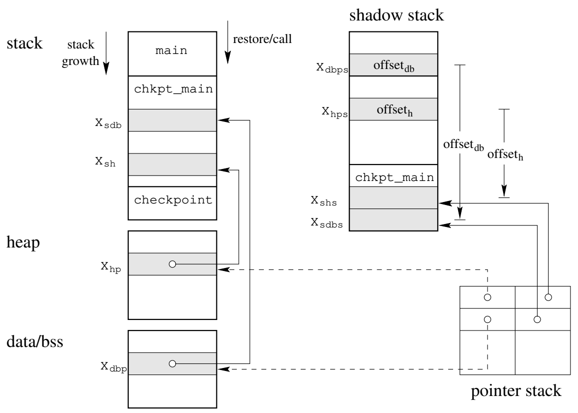


Figure 10: Recovery in the presence of stack pointers located in the heap and data/bss segments.

5.3.4 Backward Stack Pointers

As elaborated earlier, the only backward pointers that might occur on the stack are intra-frame pointers. Backward stack pointers require special treatment, but can be restricted to the case that backward pointers point to another pointer, which are rare within the scope of a function. The trick is to order variable declarations such that all non-pointer declarations precede all pointer declarations. Within a stack frame, pointers will then appear on top of all non-pointer variables. We furthermore assume that the variables of a single stack frame are saved in the reverse order than that of declaration. All pointers to non-pointer variables will then be forward pointers. Only pointers pointing to another pointer on top of itself will be backward stack pointers.

Checkpointing of backward pointers is described below, and an example shown in Figure 11, where X_p is a backward stack pointer to X.

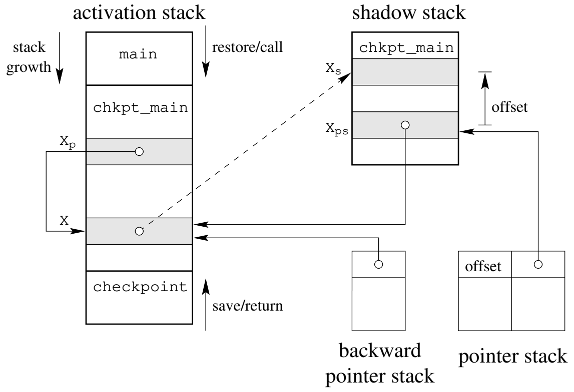


Figure 11: Checkpointing in the presence of a backward stack pointer.

Save phase: The save phase of Algorithm 1 must be preceded by a pointer collection.

> **Backward pointer collection:** Before saving the existential variables on the shadow stack, push all pointer targets of backward pointers onto a new stack, the *backward pointer stack*. In Figure 11, X, the pointer target of backward stack pointer X_p is pushed onto the backward pointer stack.
>
> **Save sweep:** Copy an object onto the shadow stack. If the object is pointed to from the backward pointer stack, i.e. it is a backward pointer target, save the mapping temporarily by overwriting the object's base address 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 is a backward stack pointer, the offset is calculated, and saved on the pointer stack. In the example, the offset is $[[X_p]-]X_{ps} = X_s - X_{ps}$. Note, that the offset of a backward stack pointer is positive.

Restore phase: The restore phase must be receded by a pointer substitution phase.

> **Restore sweep:** Forward pointers are restored as described above. Backward pointers are copied back into the activation stack, just as non-pointer objects.
>
> **Pointer substitution:** Replace all pointers on the shadow stack by their shadow stack offsets. This requires a single sweep through the pointer stack.

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.

Recovery of stack-located backward stack pointers is analogous to forward stack pointers located in data/bss or the heap. In both cases the pointer is restored before its target during recovery. Note that a backward pointer on the shadow stack can be recognized by a positive offset.

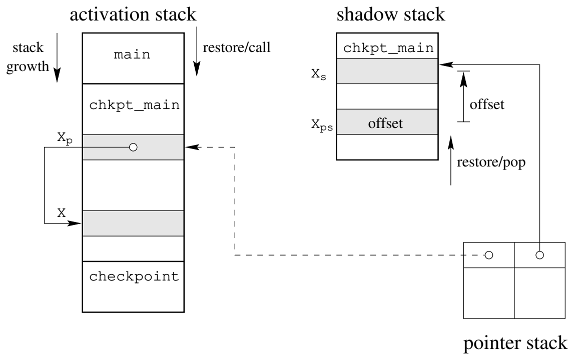


Figure 12: Recovery in the presence of a backward stack pointer.

Figure 12 illustrates the recovery of the backward stack pointer, whose checkpointing is illustrated in Figure 11. During the restore phase of recovery, the pointer stored on the shadow stack in address X_{ps} is visited before its target. A pointer to the pointer address on the activation stack X_p is pushed onto the pointer stack to fill in the actual pointer later. Also, a pointer to the pointer target shadow address X_s, computed by rule 2, is pushed onto the pointer stack. When shadow stack address X_s is visited later during the restore phase, its corresponding activation stack address X is saved in X_p.

5.4 Heap Pointers

Unlike the stack, the heap does not require restoration. Furthermore, all heap objects are aligned according the UCF convention, so that all heap offsets remain invariant. Therefore, the offset computation of heap pointers is straightforward; the offset is the heap pointer target address minus the heap bottom address. The distinction between forward and backward pointers is redundant for heap pointers.

Instead of saving heap pointers located in the heap on the shadow stack, the offset can be saved immediately.

The handling of heap pointers located in the data/bss segment and on the stack is illustrated in Figure 13.

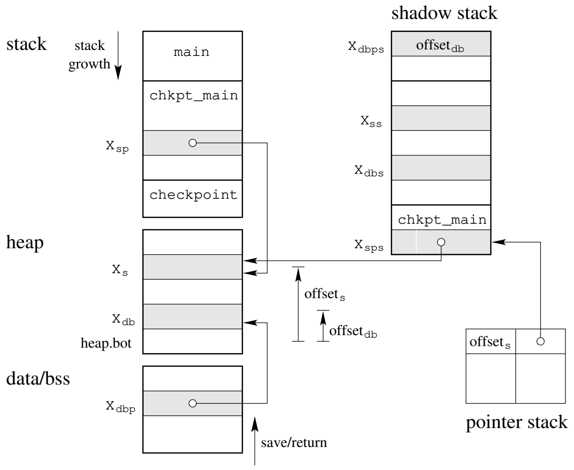


Figure 13: Checkpointing in the presence of heap pointers located on the stack and data/bss segment.

First, we consider the heap pointer in the data/bss segment, located in address X_{dbp}, and pointing to X_{db}. Since the data/bss segment is pushed onto the shadow stack before the heap, all heap pointers in data/bss are forward pointers. However, since heap pointer offsets can be computed immediatly when visiting the heap pointer, no pointer stack is required. In Figure 13, the offset becomes offset$_{db}$ $= X_{db} -$ heap.bot, and is saved in X_{dbps}.

Next, we treat the heap pointer in X_{sp} on the stack. Since the stack needs to be restored, temporary storage is required for the offset that can be calculated during the save phase. This storage is provided by the pointer stack, as described above. During the save phase in the example, the offset that replaces the heap pointer shadow in X_{sps} is computed offset$_s$ $= X_s -$ heap.bot and saved on the pointer stack. After restoring the stack, the offset replaces the copy of the heap pointer in X_{sps}.

The simplicity of the heap pointer treatment is payed for by a less space efficient heap memory management, which requires UCF compatible alignment. Since UCF specifies the largest data representations and alignments existing in the network, all UCF incompatible architectures suffer from a less space efficient data layout. We believe that this design decision is justified by the facts that architectures are converging and memory capacities are growing.

Recovery of heap pointers is simple, because the memory layout of the heap is invariant, based on UCF alignment. All heap pointers can be recovered immediately by adding the offset stored on the shadow stack to the heap bottom address.

5.5 Data/Bss Pointers

Like the heap, the data/bss segment does not require restoration, with the exception of pointer target addresses. Unlike the heap, the data/bss segment may contain data that do not require checkpointing. For example, our runtime system provided in the *shadow library*, manages several book keeping tasks in the data/bss segment, resulting in a large amount of data that are not existential. Thus, analogous to the stack, we only checkpoint the existential variables of the data/bss segment.

Since the data/bss segment is pushed onto the shadow stack before the heap and stack, all data/bss pointers located in the heap and stack are backward pointers. Since we can only collect backward pointers on the stack while saving the stack, we cannot build up a backward pointer stack before the save sweep. Consequently, the mapping from data/bss objects to their copies on the shadow stack must be saved when saving the data/bss segment. We do so by overwriting the word at the base address of the data/bss object with its shadow address (cf. broken lines in Figure 14).

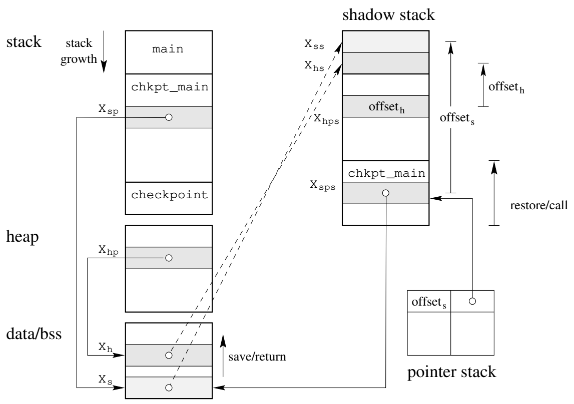


Figure 14: Checkpointing in the presence of data/bss pointers located on the stack and heap.

After the data/bss segment, the heap is saved. The data/bss pointer on the heap can be resolved immediately, because the data/bss shadow mapping is available. offset$_h$ $= [[X_{hp}]] - X_{hps}$ is saved in X_{hps} of the shadow stack.

The data/bss pointer on the stack can be resolved during the save phase, because it is a backward pointer. offset$_s$ $= [[X_{sp}]] - X_{sps}$ is pushed together with a pointer to the pointers shadow in X_{sps} onto the pointer stack. After the stack is restored, and during the pointer substitution sweep, the pointer on the shadow stack will be replaced by the offset.

Additionally, a restore phase is needed for the data/bss segment to copy the shadow values of all object base addresses back into the activation segement. Since the pointers to these values are stored in the base addresses, this substitution is straightforward; for example, $X_s \leftarrow [[X_s]]$.

Recovery in the presence of data/bss pointers located on heap or stack is analogous to the situation in Section 5.3.2. Since the data/bss segment is restored first, pointer targets in data/bss cannot be recognized while restoring this segment. Consequently, the mapping from the data/bss shadow addresses to the activation addresses needs to be stored. This is done by overwriting the shadow stack values, as indicated by the broken lines in Figure 15.

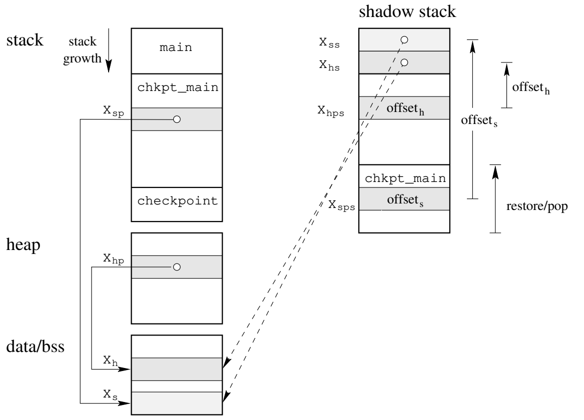


Figure 15: Recovery in the presence of data/bss pointers located on the stack and heap.

When restoring the datas/bss pointers on the heap and stack, these pointers can be computed by means of the lookup operation: $X_{hp} \leftarrow [X_{hps} + [X_{hps}]]$ and $X_{sp} \leftarrow [X_{sps} + [X_{sps}]]$.

During checkpointing, backward data/bss pointers located in the data/bss segment can be resolved immediately, because the target's shadow stack address can be found in the pointer target address, as indicated by the broken lines in Figure 14. Data/bss-located forward data/bss pointers, however, require the use of the pointer stack to save the pointer's shadow address until the the pointer target's shadow address is known. Figure 16 illustrates this case. When the pointer target X is saved, the offset can be computed and stored in X_{ps} immediately.

Recovery of data/bss-located data/bss pointers requires to distinguish between forward and backward pointers. With forward pointers, the pointer will be restored before its target. Consequently, the pointer stack is required to save the pointer temporarily. This situation resembles the one in Figure 12. For backward pointers the scheme explained above (Figure 15) can be applied.

5.6 Optimizations

The separation of checkpointing into two phases offers an advantage for handling pointers: During the save phase,

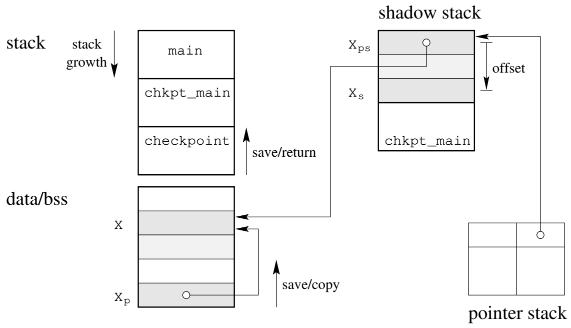


Figure 16: Checkpointing in the presence of forward pointers within the data/bss segment. Forward refers to the direction of the save traversal.

we can gather *all* pointers on the pointer stack. Maintaining the pointer stack as a sorted list reduces the complexity of checking the pointer stack (item 4(1) in Algorithm 3) from $O(n)$ to $O(1)$. An $O(n \log n)$ sorting algorithm reduces the overall overhead for m potential pointer targets from $O(n * m)$ to $O(n \log n + m)$. Unlike inter-frame pointers, intra-frame stack pointers can be resolved during the save phase. Introducing a separate "resolved pointer stack" to store offsets that can be computed during the save phase will improve performance by reducing n.

To reduce the overhead caused by forward pointers, the declaration list of variables can be reordered such that all pointers are declared after non-pointer variables. As a result, all intra-frame stack pointers and data/bss pointers to non-pointer targets will become backward pointers.

Further optimizations are also possible when checkpointing heap memory. Since memory management is performed by the runtime system, separate used and free lists of allocated blocks are maintained. Only blocks on the used list are currently checkpointed. This, however, may include unreachable memory that has been allocated but not freed. It may be cost effective to perform garbage collection [1] before checkpointing the heap to further reduce the amount of information that needs to be checkpointed.

6 Experimental Results

Two types of experiments were performed to evaluate the performance of portable checkpoints: (1) Microbenchmarks to shed light on the cause of overheads, and (2) three small application programs to demonstrate the runtime efficiency of our implementation techniques, in particular checkpointing overhead and performance in the presence of failures.

6.1 Microbenchmarks

We use three simple programs to analyze the overhead induced by portable checkpoints: The recursive formulation of the Fibonacci number computation shown in Figure 4 consists almost only of checkpointing instrumentation overhead. A simple version of the C-library function `memcpy` demonstrates the runtime penalty, and a modified version `structcpy` illustrates the overhead due to UCF alignment. All runtimes reported are average values of five measurements.

6.1.1 Code Instrumentation Penalty

The transformation of the Fibonacci program in Figure 3 into the code in Figure 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 stackframes 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.

Table 1 shows `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 example. 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.

6.1.2 Shadow Stack Overhead

The C-library routine `memcpy` copies the contents of an array bytewise into another. In this example we allocate two arrays on the heap, and pass pointers to their base addresses to the `memcpy` routine, analogous to the code in Figure 6. Within this routine is a loop, that copies the array contents bytewise. We specified a potential checkpoint location within this loop.

The size of the two arrays is 10^6 bytes. Thus, the checkpoint function is entered 10^6 times. Since two large arrays are checkpointed entirely and the function calling sequence is only 2 deep, the checkpointing overhead is dominated by copying these arrays. Thus, this benchmark complements the measurement of the stack induced overhead of the Fibonacci program. Checkpoint size and break-down for `memcpy` are the same than for Sunos in Table 3. Table 2 shows the runtimes of different systems without checkpoint instrumentation (*plain*), with instrumentation but without storing a single checkpoint (*instr*), saving *one* checkpoint — specified by using an appropriate timer value — on the shadow stack without writing it to disk (*copy*), saving *one* checkpoint on the shadow stack and writing it to a local disk (*local*) and via NFS to a remote disk (*NFS*).

System	plain	instr	copy	local	NFS
i486/Linux	0.71	1.09	2.26	6.54	—
Sparc1+/4.1	0.59	1.78	2.43	5.78	21.76
Sparc20/5.3	0.10	0.46	0.66	0.79	8.56

Table 2: `memcpy` runtimes in seconds.

The overhead of code intrumentation is the difference *instr−plain*. The cost of saving a single checkpoint on the shadow stack, including save and restore phase is the difference *copy−instr*. The time to store a single checkpoint to local or remote disk is *local−copy* or *NFS−copy*, respectively. The Linux PC was not connected to NFS for these experiments.

As expected, transferring the checkpoint to remote disk is the most expensive portion of checkpointing that determines the MinTBC eventually. Furthermore, the instrumentation overhead is not negligible, suggesting that potential checkpointing locations should be chosen carefully. The copying overhead onto the shadow stack is tolerable, because it enables us to hide the more time consuming transfer to disk with useful computation.

6.1.3 UCF Overhead

Program `structcpy` resembles `memcpy`, but copies an array of structures elementwise. To demonstrate the effect of alignment, we use the `cd_t` structure, shown in Figure 2. Running the `structcpy` example with arrays of $62,500$ elements and without padding for UCF alignment yields the checkpoint sizes in Table 3.

If the i486/Linux system is operated with a UCF that matches the SPARCstations, the padding suggested in Figure 2 is introduced. Then, the checkpoint size on the i486/Linux system becomes that of the Sparc/Sunos systems, increasing the overhead of checkpointing. This alignment penalty is seen in Table 4, where the i486/non-UCF runtimes correspond to the Linux checkpoint size in Table 3, and the i486/UCF runtimes to the Sunos checkpoint size.

	Linux	Sunos
sizeof(cd_t)	12	16
file	1,500,252	2,000,252
data/bss	108	108
heap	1,500,052	2,000,052
stack	28	28

Table 3: Space requirements of `structcpy` without padding.

System	plain	instr	copy	local	NFS
i486/non-UCF	0.12	0.21	1.17	3.86	—
i486/UCF	0.14	0.29	1.48	6.75	—
Sparc1+/4.1	0.21	0.29	0.96	4.38	23.13
Sparc20/5.3	0.04	0.06	0.26	0.41	8.58

Table 4: `structcpy` runtimes in seconds.

The runtimes presented here are comparable to the `memcpy` times, because the UCF compatible arrays have the same size. Obviously, elementwise copying is faster than bytewise copying.

6.2 Applications

We use three applications to measure the overhead of checkpointing. Two applications, a Jacobi-type iteration and a matrix multiplication are floating point intensive and operate on large data sets, resulting in large checkpoints. The third application is a recursive depth-first search of prime numbers, which generates a deep stack hierarchy, where the stack save and recovery phases dominate the checkpointing overhead. Furthermore, the Jacobi-type iteration is also used to measure the lower bound of the runtime of the program in the presence of failures.

	heat	matmult	prime
file	1,061,208	4,546,420	10,836
data/bss	2,156	116	10,528
heap	1,058,884	4,546,148	20
stack	104	92	224

Table 5: Checkpoint sizes and break-downs for example applications in bytes.

Table 5 reveals the amount of data stored in the checkpoints. The difference between checkpoint file sizes and the sum of the corresponding segment sizes is the checkpoint header size (64 bytes). No dynamic variables are allocated in `prime`; the 20 bytes of the heap segment contain memory management information.

All experiments are performed with *sequential checkpointing*, where the program is halted to copy its state onto the shadow stack, then writes the checkpoint to disk, and, after completion, continues the program execution. Optimizations, such as proposed in [6, 17], can be applied.

6.2.1 Heat Equation

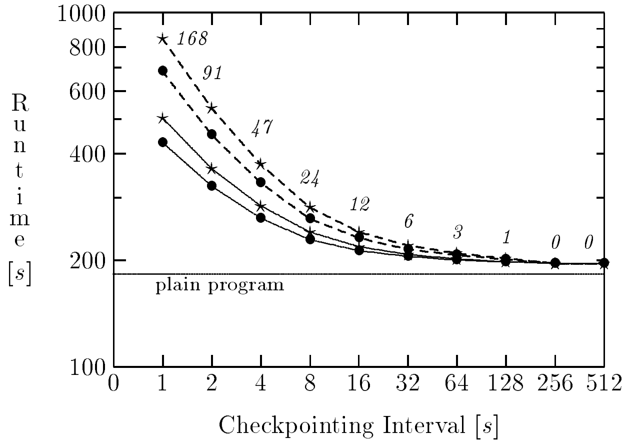
We use a Jacobi-type iteration to solve the heat diffusion problem on a 256×256 grid, executing $1,000$ iterations. Two dynamically allocated two-dimensional `double` arrays are used, one to hold the temperature values of the current iteration, the other to store the results of the five-point-stencil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

The potential checkpoint location is placed within the outer iteration loop. It is thus visited $1,000$ times. Figure 17 summarizes the results of our experiments with checkpointing to the local disk. We measured the runtimes for a range of MinTBC (timer value) between 0 and 512 seconds. For MinTBC$=0$, each potential checkpoint location is activated. The graphs in Figure 17 plot the runtimes for UCF compatible and UCF incompatible checkpointing. Checkpointing is UCF compatible, if the UCF specification matches the system architecture, so that no conversion is required. With UCF incompatible checkpointing, alignments and conversions are performed on the i486 to match the format of the SPARCstations and vice versa. The node attributes give the number of checkpoints saved for the corresponding MinTBC.

Figure 17 illustrates how often checkpoints can be saved without affecting performance substantially. On all systems, a checkpointing interval (MinTBC) larger than 32 seconds restricts the overhead to less than $10\,\%$. Although this value depends on the checkpoint size, it is small compared to typical system MTBF values. Note that the conversion penalties paid for UCF incompatibility are only severe, if the checkpointing frequency becomes unrealistically high, i.e. MinTBC very small.

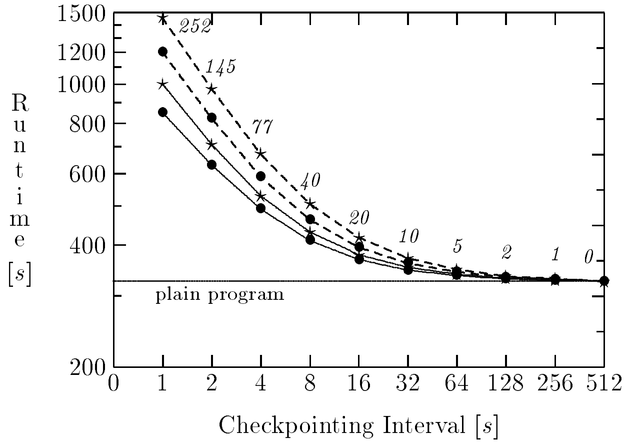
Columns labeled t_{rec} in Figure 17 give the minimum run times of the program, if failures occur approximately every MinTBC. This "ideal" failure situation is simulated by exiting the program just after a checkpoint has been stored, capturing the exit status within a shell script that immediately invokes the program again with the recover option enabled. Since the program is aborted immediately after a checkpoint is stored, no recomputation 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, about $1.5\,s$ on the SPARC-



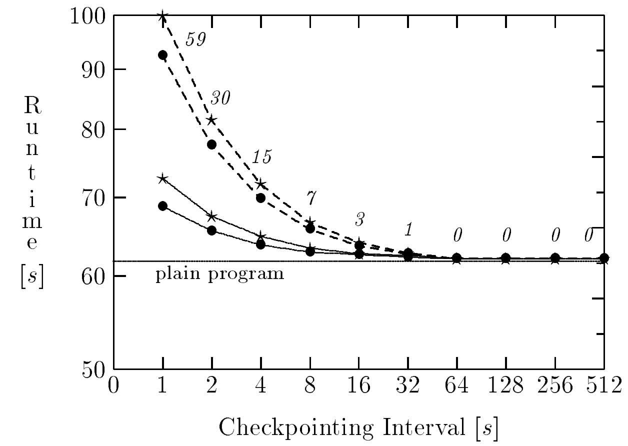
# of chkpts	UCF compatible			UCF incompatible		
	t_{chkpt}	ovh	t_{rec}	t_{chkpt}	ovh	t_{rec}
0	196.1	7	196.1	196.1	7	196.1
1	198.0	8	201.3	198.0	8	202.6
3	200.4	9	207.2	201.9	10	209.5
6	205.2	12	215.0	207.8	13	220.5
12	213.0	16	231.6	218.7	19	240.7
24	228.9	25	262.3	240.5	31	282.7
47	262.9	44	331.6	285.2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

(a) IBM Thinkpad i486DX475 / Linux



# of chkpts	UCF compatible			UCF incompatible		
	t_{chkpt}	ovh	t_{rec}	t_{chkpt}	ovh	t_{rec}
0	325.8	0	325.8	325.8	0	325.8
1	328.2	1	330.0	328.6	1	330.8
2	330.3	1	333.5	331.4	2	335.4
5	336.8	3	343.9	339.3	4	348.9
10	347.1	7	361.2	352.8	8	371.9
20	368.6	13	395.6	379.0	16	418.5
40	410.9	26	463.1	431.9	33	507.0
77	491.7	51	590.6	529.7	63	672.9
145	631.4	94	825.7	709.7	118	975.8
252	851.1	161	1201.6	1001.2	207	1460.8
1000	2441.6	649	—	2977.7	813	—
runtime without instrumentation: 325.8 s						

(b) Sun SPARCstation1+ / SunOS4.1 (BSD)



# of chkpts	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
1	62.3	1	62.7	62.5	1	62.8
3	62.6	1	63.7	62.7	1	64.1
7	62.9	2	65.8	63.4	3	66.7
15	63.8	3	69.9	64.9	5	71.9
30	65.6	6	77.6	67.5	9	81.5
59	68.8	11	92.5	72.7	18	100.2
1000	173.7	181	—	241.0	290	—
runtime without instrumentation: 61.8 s						

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

Figure 17: Heat equation on three different systems, storing UCF checkpoints on the local disk. Runtimes t_{chkpt} without failures and t_{rec} in the presence of failures are given in seconds, the overhead of checkpointing (ovh) in per cent with respect to the runtime without instrumentation.

System	MinTBC	# of	without storing checkpoint to disk				storing checkpoint to local disk			
			UCF compatible		UCF incompatible		UCF compatible		UCF incompatible	
	[s]	chkpts	time	ovh	time	ovh	time	ovh	time	ovh
i486/	32	5	456.2	154	443.0	147	565.5	215	566.5	216
Linux	64	2	272.6	52	274.1	53	328.4	83	331.5	85
	128	1	215.0	20	218.5	22	235.1	31	235.4	31
(179.4 s)	256	0	180.5	1	181.7	1	181.5	1	183.3	2
	32	9	314.7	5	332.6	11	382.7	27	412.4	37
Sparc1+/	64	4	307.8	2	317.1	6	353.9	18	356.1	19
Sunos4.1	128	2	306.0	2	311.5	4	330.2	10	340.5	13
	256	1	304.5	1	307.6	2	319.7	6	323.6	8
(300.3 s)	512	0	302.9	1	303.8	1	303.3	1	302.6	1
			storing checkpoint to local disk				storing checkpoint via NFS			
	0	615	351.5	460	562.9	796	—	—	—	—
	1	61	91.5	46	112.0	78	1313.8	1992	1203.6	1817
	2	30	77.0	23	89.1	42	719.2	1045	632.3	907
Sparc20/	4	15	70.1	12	75.1	20	328.7	423	327.9	422
Sunos5.3	8	7	67.6	8	68.7	9	185.7	196	202.3	222
	16	3	64.5	3	65.9	4	115.4	84	119.2	90
(62.8 s)	32	1	63.5	1	63.8	2	79.6	27	79.1	26
	64	0	62.8	0	62.8	0	62.8	0	62.8	0

Table 6: Dense matrix-matrix multiplication (615×615). Runtimes without instrumentation are given in parentheses in the system column. Runtimes are given in seconds, overheads in per cent relative to the runtimes without instrumentation.

station1+, and $0.4\,s$ on the SPARCstation20. These numbers are determined by the use of the local disk as stable storage for the checkpoint. All 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.

6.2.2 Matrix Multiplication

We instrumented the dense matrix-matrix multiplication implemented by Plank [17]. Table 6 summarizes the runtimes and overheads for the checkpointed dense matrix-matrix multiplications of two dense 615×615 matrices without failures.

The performance of the i486 is dominated by its local disk performance. Data conversion overhead of the UCF incompatible run times is submerged by the variance of the disk performance, which is primarily caused by swap activity. The $8\,$MB of main memory cannot quite hold Linux plus two copies of the three matrices, one on the activation heap and the other on the shadow stack. Consequently, even a single checkpoint adds a significant overhead of 20 – $30\,\%$. This leads to two conclusions: (1) Avoid swapping by choosing the problem size such that activation segments as well as shadow stack fit into main memory. As future memory capacities are growing, it is more likely that cache effects will determine performance. (2) Dense matrix-matrix multiplication is a basic linear algebra operation that hardly appears stand-alone, but is usually embedded within a larger application. Obviously, it is more sensible to checkpoint after the multiplication rather than within. Such observations raise interesting incentives for compiler data flow analysis to identify potential checkpoint locations automatically.

The SPARCstation1+ measurements show that data conversion adds a significant overhead of approximately $5\,\%$ compared to the UCF compatible runtimes. Storing checkpoints to the local disk introduces an overhead that depends on the capability of the memory buffers for disk access and the speed of the disk. Obviously, the i486 system delivers a substantially better local disk performance than the SPARCstation1+. When storing the checkpoint to remote disk, as presented for the SPARCstation20, the overhead increases dramatically, as already shown with the `memcpy` micro-benchmark.

MinTBC [s]	i486/Linux # of chkpts	time [s]	ovh [%]	Sparc1+/Sunos4.1 # of chkpts	time [s]	ovh [%]	Sparc20/Sunos5.3 # of chkpts	time [s]	ovh [%]
0	256	48.8	87	256	294.9	18	256	83.6	1.3
1	24	28.6	10	188	282.7	13	69	82.8	0.4
2	12	27.4	5	101	267.9	7	37	82.7	0.2
4	6	26.9	3	55	260.1	4	19	82.6	0.1
8	3	26.6	2	29	255.6	2	10	82.6	0.1
16	1	26.4	1	15	253.3	1	5	82.5	0.0
32	0	26.3	1	7	251.9	1	2	82.5	0.0
64				3	251.1	0	1	82.5	0.0
128				1	250.7	0	0	82.5	0.0
256				0	250.5	0			
plain	—	26.1	—	—	250.5	—	—	82.5	—

Table 7: Prime runtimes and checkpointing overhead, storing checkpoints to local disk.

6.2.3 Prime

The prime benchmark uses a recursive divide-and-conquer algorithm to compute the primes among all natural numbers less than a user-specified upper bound. The user also specifies a grain size which determines the depth of the divide-and-conquer tree. The range 2-upper bound is recursively partitioned into two equal halves until each partition is less than or equal to the grain size. An optimized Eratosthenes sieve algorithm is used on each partition to determine which numbers in that range are prime. The conquer phase is used to compute the total number of primes found.

Table 7 contains the results of running prime without failures on the first 10,000,000 natural numbers with a grain size of 250. The last line in the table provides the reference measurement without any checkpointing or instrumentation overhead. All overheads are reported relative to this case. Note that when checkpoints are taken every 2 seconds, the overhead on all three machines is less than 7 %. Although not reported here, the data for writing to a remote disk via NFS yields marginally higher overheads for checkpoints taken every 2 seconds or greater. This is not surprising since the size of each checkpoint is relatively small with less than 11K bytes (see Table 5).[4]

7 Conclusion

We have introduced the concept of portable checkpoints, and presented the runtime and compiler support needed to implement portable checkpoints. Furthermore, we have demonstrated that the overhead introduced by portable checkpointing is very low when reasonable checkpoint intervals are chosen, even without hiding the latency of transferring the checkpoint to remote storage. For programs with large checkpoints such as heat or matrix multiplication, network/disk performance is the primary obstacle, compared to which the overhead of saving a checkpoint on the shadow stack is negligible.

A universal checkpoint format (UCF) that permits checkpoints to be ported across UCF-compatible and UCF-incompatible systems has been developed. The overhead of converting checkpoints into a UCF-compatible format on non-UCF machines was found to be negligible except when the frequency of checkpointing was unrealistically high. Checkpoint portability was validated on the three systems reported by checkpointing the program on one system, transferring the checkpoint onto a different system, and successfully resuming the execution there.

The cost of recovery from failures in our scheme was found to be very low on UCF-compatible machines, and not surprisingly, a little higher on UCF-incompatible machines. Our experiments show that the total volume of data that needs to be recovered is the determining factor in recovery cost; the system overhead is very small.

The instrumented versions of the benchmark programs were hand-translated to obtain the data reported in this paper. We are currently implementing a source-to-source C compiler that performs the program transformations described in this paper. We expect to incorporate additional optimizations to further reduce the amount of data that needs to be checkpointed, and introduce latency hiding support at run time to reduce the cost of writing a checkpoint to remote disk.

The proposed scheme only requires that (1) a user program be submitted to a front-end source-to-source C com-

[4] Note that the prime example of evolutionary computer architecture, the i486, is up to three times faster than a SPARCstation 20, and almost 10 times faster than a SPARCstation1+.

piler before compilation on the desired target machine, and (2) the 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 any C program robust in the presence of faults and recoverable on any UNIX-based system.

Acknowledgment

We are grateful to Jon Kuhl for his valuable suggestions to improve the quality of this manuscript.

References

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4), June 1987.

[2] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995.

[3] Nicholas S. Bowen and Dhiraj K. Pradhan. Virtual checkpoints: Architecture and performance. *IEEE Transactions on Computers*, 41(5):516–525, May 1992.

[4] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3:63–75, 1985.

[5] Geert Deconinck, editor. *Integrating Fault Tolerance in off-the-shelf Massively Parallel Systems*, Workshop run by FTMPS consortium (ESPRIT 6731) within HPCN Europe 1996, 1996.

[6] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, Houston, Texas, October 1992. IEEE.

[7] Michael Franz. *Code Generation On the Fly: A Key for Portable Software*. PhD thesis, Institute for Computer Systems, ETH Zürich, 1994.

[8] James Gosling and Henry McGilton. *The Java Language Environment, A White Paper*. Mountain View, CA, May 1995.

[9] David B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *12th Symposium on Reliable Distributed Systems*, Princeton, New Jersey, October 1993. IEEE.

[10] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum. Transparent Fault-Tolerance in Parallel Orca Programs. In *Symposium on Experiences with Distributed and Multiprocesor Systems*, pages 297–311, March 1992.

[11] R. Koo and Sam Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Enigeering*, SE-13 no. 1:23–31, January 1987.

[12] Chung-Chi Jim Li, Elliot M. Stewart, and W. Kent Fuchs. Compiler-assisted Full Checkpointing. *Software — Practice and Experience*, 24(10):871–886, October 1994.

[13] Kai Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

[14] B. Lyon. *Sun External Data Representation Specification*. Sun Microsystems, Inc., Mountain View, 1984.

[15] N. Neves, M. Castro, and P. Guedes. A Checkpoint Protocol for an Entry-Consistent Shared Memory System. In *Proceedings of Symposium on Principles of Distributed Computing*, pages 121–129, August 1994.

[16] James S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Princeton University, June 1993.

[17] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent checkpointing under Unix. In *USENIX Winter 1995 Technical Conference*, pages 213–233, New Orleans, Louisiana, January 1995.

[18] Daniel J. Scales and Monica S. Lam. Transparent Fault Tolerance for Parallel Applications on Networks of Workstations. In *Usenix Winter Conference*, January 1996.

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

[20] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely asynchronous optimistic recovery with minimal rollbacks. In *25th Annual International Symposium on Fault-Tolerant Computing - Digest of Papers*, Pasadena, CA, June 1995. IEEE Computer Society.

[21] Mark E. Staknis. Sheaved memory: Architectural support for state saving and restoration in paged systems. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–102, Boston, Massachusetts, April 1989.

[22] Volker Strumpen. Software-based communication latency hiding for commodity workstation networks. To appear in International Conference on Parallel Processing, August 1996.

[23] Alan Wood. An analysis of client/server outage data. *IEEE*, pages 295–304, 1995.