Compiler Technology for Portable Checkpoints

Volker Strumpen*
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
strumpen@theory.lcs.mit.edu

Abstract

We have implemented a prototype compiler called porch that transforms C programs into C programs supporting portable checkpoints. Portable checkpoints capture the state of a computation in a machine-independent format that allows the transfer of computations across binary incompatible machines. We introduce source-to-source compilation techniques for generating code to save and recover from such portable checkpoints automatically. These techniques instrument a program with code that maps the state of a computation into a machine-independent representation and vice versa. In particular, the following problems are addressed: (1) providing stack environment portability, (2) enabling conversion of complex data types, and (3) rendering pointers portable. Experimental results show that the overhead of checkpointing is reasonably small, even if data representation conversion is required for portability.

1 Introduction

This paper presents a source-to-souce compiler technique that transforms sequential programs into semantically equivalent source programs which are additionally capable of saving and recovering from portable checkpoints across binary incompatible machines. This technique allows a source-to-souce compiler to assume a major portion of the work necessary for providing process migration and fault tolerance in heterogeneous environments.

Our prototype compiler porch transforms C programs written in a portable manner, i.e., C programs that compile and run on different architectures of a computer network without modifications to the source code. If such a C program is compiled with porch, a new C program is generated that contains the code for checkpointing and recovery, as well as code to convert data representations if necessary. This C program can be compiled with any native ANSI C compiler to produce an executable for a particular target architecture. During execution, checkpoints may be saved which are machine-independent, allowing a binary incompatible machine to recover the computation. The porch compiler is itself written in C using the literate programming tool `noweb`. The C language has been chosen for the porch prototype, because its imperative features such as pointers, type casting, or union data types allow for exploring the limitations of the source-to-source compilation approach.

The porch compiler resolves three major obstacles to portability: the stack environment is system-specific, the layout of complex data types is architecture-specific, and pointers appear to be inherently nonportable. The *stack environment* exhibits a variety of system-specific issues including hardware support for function calls, register file layout, hidden state such as the stack pointer, the frame layout, which is determined by a particular compiler, and operating system compliance. By visiting each individual stack frame, all local variables can be accessed by their name to save or recover them from a checkpoint. The second obstacle to portability are architecture-specific data representations and layout of complex data types. The latter is determined by both architecture and compiler. We introduce a structure metric that provides a specification of the data layout at runtime to accomplish *data representation conversion*. The third portability hazard is language specific: At the first glance it seems impossible to provide *portability of pointers* in imperative languages like C. We show, however, that pointers can be translated into machine-independent offsets within the checkpoint to render them portable.

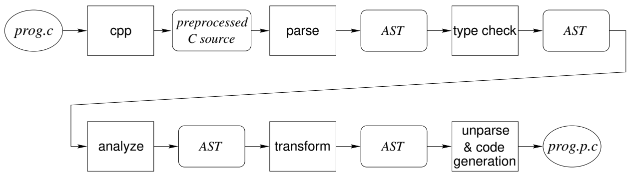


Figure 1: Translation phases of porch.

The porch source-to-source compiler consists of the modules shown in Figure 1. The input to porch is a C program consisting of one or multiple C files. Not all files of a program need to be input to porch; those not given will not be instrumented. Figure 1 illustrates the transformation of input file *prog.c* into file *prog.p.c*. The internal representation is a forest of abstract syntax trees (ASTs), generated by the parser, and subsequently type checked. The analysis phase includes interprocedural call graph analysis and, optionally, live variable analysis. AST-based code transformations assist the checkpointing process, including moving variable initializers out of declarations, moving certain function calls out of expressions, or assigning constant strings to global variables. The generation of checkpointing and recovery code is mostly integrated with the unparse phase, because it merely involves adding code.

The performance of porch has not been the primary issue during its design. The compilation phases, including type checking, analysis, transformations, and checkpoint-code generation

*This work was supported in part by DARPA Grant N00014-94-1-0985 as part of the Cilk Project. Beta-release available at *http://theory.lcs.mit.edu/˜porch*

are mutually independent. Nevertheless, experience shows that porchifying a program is usually faster than the subsequent native compilation and link steps. Our performance studies have also shown that the overhead of compiler generated checkpointing is limited to few percent of the runtime for reasonable checkpointing frequencies, even in the presence of data representation conversions for portability [10, 15].

Besides removing the burden of coping with heterogeneity of computer architectures, the proposed technology is well suited for a variety of objectives. porch can be used to provide functionalities such as fault-tolerance, process migration in heterogeneous environments, process migration after recompilation, or load balancing. Portable checkpoints ease retrospective diagnostics, because the checkpoint can be inspected on a binary incompatible machine within an established software environment, for instance to assist new hardware developments. Furthermore, porch simplifies implementing system functionalities such as compiler-generated message marshalling or heterogeneous distributed shared memory. Here, the runtime information supported by porch could be used to reallocate heap objects to avoid false sharing, for example. In short, we view porch as a step towards solving well defined parts of the increasing complexity of programming heterogeneous systems automatically.

We focus our discussion on the three key techniques implemented in porch: stack portability, data representation conversion and pointer portability. This paper does not attempt to solve the many idiosyncracies involved in implementing a production compiler. Section 2 introduces our concept of portable checkpoints. In Section 3, we describe a general method for saving and recovering the stack in a machine-independent manner. A technique for data representation conversion is presented in Section 4, including the treatment of unions. Section 5 addresses the problem of pointer portability; it discusses how pointers can be converted into machine independent offsets. Section 6 presents the handling of dynamically allocated memory. Section 7 summarizes the runtime algorithms for checkpointing and recovery. Experimental results evaluate the checkpointing and recovery performance in the presence of pointers in Section 8. Related work is sketched in Section 9.

2 Portable Checkpoints

This section formalizes our notion of a portable checkpoint. As the following definition shows, we view a portable checkpoint as abstract data type.

Given a program $\mathcal{P}$ and a finite set of locations $k \in \mathcal{L} = \{1, \ldots, n\}$, specified in the program text, which are called ***potential checkpoint locations***.[1] Furthermore, given a finite set of machine formats $\mathcal{M} = \{UCF, M_1, \ldots, M_m\}$ describing the data representations of basic data types. The format ***UCF*** is a special parameterizable machine format, called the ***Universal Checkpoint Format***, in which checkpoints are represented. A program state at location k in machine format M_i during execution is denoted $\mathcal{S}^k_{M_i}$.

Definition 1 A ***portable checkpoint*** is the state of a computation $\mathcal{C}^k_{UCF}$ that exists, if for all machine formats M_i and M_j the following pair of operations exist:

$$\begin{aligned} \texttt{checkpoint}^k_i: &\quad \mathcal{S}^k_{M_i} \rightarrow \mathcal{C}^k_{UCF}, \\ \texttt{recover}^k_j: &\quad \mathcal{C}^k_{UCF} \rightarrow \mathcal{S}^k_{M_j}. \end{aligned}$$

The technology proposed in this paper is a set of compiler transformations that can be viewed as a function $\mathcal{T}$:

$$\begin{aligned} \mathcal{T}:\ \ \mathcal{P} + \mathcal{L} + M_i + UCF \quad &\rightarrow \\ \mathcal{Q} + \{(\texttt{checkpoint}^k_i, \texttt{recover}^k_i) \mid k \in \mathcal{L}\}. &\qquad (\dagger) \end{aligned}$$

Given a program $\mathcal{P}$ with a set of potential checkpoint locations $\mathcal{L}$, a target machine format M_i and a *UCF* specification, the function $\mathcal{T}$ generates program $\mathcal{Q}$ and one pair of `checkpoint` and `recover` operations for each potential checkpoint location $k \in \mathcal{L}$. Here, $\mathcal{Q}$ is semantically equivalent to $\mathcal{P}$, but is instrumented to provide additional functionality to enable saving and recovering from portable checkpoints at all potential checkpoint locations $k \in \mathcal{L}$. The existence of portable checkpoints is independent of the choice of $\mathcal{L}$, and follows from the construction of the compiler transformations introduced in this article.

A potential checkpoint location may be enabled at runtime, depending on system-based criteria such as timers [7], or by sending a signal to the process. Upon expiration of the timer or receipt of the signal, the next visited potential checkpoint location becomes active, and a checkpoint will be saved.

The machine formats M_i, including *UCF*, describe size, alignment, byte order, and floating-point representations of all basic data types. *UCF* is a machine format, but is not bound to an individual machine; such a machine may not even exist. In practice, however, the *UCF* specification is chosen such that the `checkpoint` and `recover` operations preserve the accuracy of basic data types, minimize the conversion overhead, or reduce the checkpoint size while sacrificing accuracy.

3 Stack Environment Portability

In the following, we explain our technique for checkpointing and recovering the runtime stack in a portable manner. Source-to-source compilation is the key to this end.

For most programming languages, the runtime stack is inherently nonportable, because it reflects the hardware design of the CPU, including register set and architectural support for function calls. Furthermore, hidden registers hold values such as the program counter, stack pointer, etc. Some compilers do not reserve stack space for variables stored in registers. It is not clear whether a system-based approach without compiler support is powerful enough to save and restore the stack environment across machines with different register sets. Even with compiler support the question is how to save the state of the runtime stack without requiring knowledge of the design and layout of the stack on each of the target architectures.

The key idea is to access all stack-allocated variables by their *names* when saving and recovering from checkpoints. Register allocation and stack layout are not an issue at the source-code level, where porch operates.

Accessing variables by means of their names requires entering their lexical scope. To checkpoint the local variables of functions on the runtime stack, we start with the currently active function on top of the stack, save its local variables, and recursively visit its caller function until the bottom of the stack is reached. During recovery, the process is reversed. A function frame is pushed onto the stack, and its local variables are loaded from the checkpoint. Then, the callee of the original call sequence is pushed onto the stack until the runtime stack is rebuilt.

The question is, how can the control flow be redirected during checkpointing to enable access to local variables by their names. The only portable mechanism to visit stack frames, available on every general purpose processor architecture, is the standard function call and return. This mechanism is henceforth used to instrument functions in order to provide for extraordinary returns, called ***releases***, and extraordinary calls, called ***resumes***. A stack frame is

[1] *Potential checkpoint locations* are specified by inserting a function call to library routine `checkpoint` into the program.

released by remembering the ***resume-point*** and returning (standard function return) to the caller. A stack frame is resumed by calling the function (standard function call), and jumping to the resume-point without executing any of the function's original statements. Two code constructs implement the release and resume functionalities, a "jump table" and "call wrappers."

Definition 2 A ***jump table*** implements a *computed goto*. It consists of a switch of *goto* statements, one of which is selected by the resume-point identifier.

Definition 3 A function ***call wrapper*** consists of two parts, the ***pre-call*** code and the ***post-call*** code. The pre-call code consists of an assignment to a local state variable `callid`, which identifies the resume-point, and a subsequent label. The post-call code contains a conditional return statement.

Jump table and call wrappers constitute the "release-and-resume instrumentation" which is a machine-independent part of function $\mathcal{T}$ (†):

Transformation 1 (*Release-and-Resume Instrumentation*)
Every function on a call path to a potential checkpoint location is subject to the following code transformation:

1. *Introduce a new local variable (`callid`) to store the resume-point.*
2. *Insert a jump table before the first instruction of the function with entries corresponding to each of those function calls within this function that lead to a potential checkpoint location, including potential checkpoint locations.*
3. *Insert call wrappers around all those function calls within the function that lead to a potential checkpoint location, including potential checkpoint locations.*

We illustrate the release-and-resume instrumentation by means of an example. Function `foo` below calls function `bar` and contains a potential checkpoint location, specified by a call to library function `checkpoint`. Function `bar` is assumed to contain another potential checkpoint location.

```
void foo(void)
{
     :
   bar();
     :
   checkpoint();  /* potential checkpoint location */
     :
}
```

Figure 2 contains a simplified version of the release-and-resume instrumentation, which emphasizes the control flow redirection. The jump table is inserted before the first statement of `foo`. Both function calls `bar()` and `checkpoint()` are instrumented with a call wrapper.

The following execution scenario illustrates the control-flow redirection: During *normal execution*, flags `checkpointing` and `restoring` are false. Upon entering `foo`, the jump table is skipped. Assume that normal execution arrives at the pre-call code of `checkpoint`, when a checkpointing signal is received. In the pre-call code `callid` is set to 1, which identifies the resume-point. Then, `checkpoint` is called. Within `checkpoint`, flag `checkpointing` is lit as a consequence of having received the checkpointing signal. Upon return from `checkpoint`, the post-call conditional is true, which causes a release of the stack frame of `foo` by returning to the caller. The release of stack frames continues until the library supplied main function at the bottom of the runtime stack is reached. There, flag `checkpointing` is reset and

```
void foo(void)
{
   unsigned long callid;

   if ( restoring ) {           /* jump table     */
     switch(callid) {
       case(0): goto L_call0;
       case(1): goto L_call1;
     }
   }
       :
   callid = 0;                  /* pre-call       */
 L_call0:                       /* resume-point 0 */
   bar();                       /* call 0         */
   if ( checkpointing )         /* post-call      */
     return;
       :
   callid = 1;                  /* pre-call       */
 L_call1:                       /* resume-point 1 */
   checkpoint();                /* call 1         */
   if ( checkpointing )         /* post-call      */
     return;
       :
}
```

Figure 2: Simplified illustration of the release-and-resume instrumentation, consisting of a *jump table* at the function entry and *call wrappers* around function calls.

flag `restoring` is lit. Then, the stack is restored by resuming the same function call sequence that has been active when calling function `checkpoint`. When resuming function `foo`, resume-point identifier 1 is loaded into variable `callid`, enabling the jump table to redirect control to the pre-call of `checkpoint`. Function `checkpoint` resets flag `restoring` and returns. Now, back to normal execution, the post-call conditional is false, and the statement following the post-call conditional is executed.

In the previous discussion, we omitted how resume-point identifier 1 is assigned to `callid` when function `foo` is resumed. During checkpointing, all local variables are saved before the releasing return statement in the post-call code. Similarly, restoring local variables is integrated with the jump table. As all other local variables, `callid` is saved and restored. The instrumentation of save and restore code consists of push and pop operations on a ***shadow stack***, whereby local variables are accessed by name. A description of the save and restore code generation can be found in [10]. Note that recovery involves no more than resuming the functions on the runtime stack.

Several code transformations are required prior to the release-and-resume instrumentation that are not elaborated here. Among the obvious transformations are moving initializers beneath the jump table, moving function calls on a call path to a potential checkpoint location out of expressions, introduce dummy return values for the releasing return, and moving declarations in nested blocks to the top level to unify the name space of local variables. Furthermore, optional live-variable analysis may be performed to identify those local variables for which checkpointing and recovery code must be generated.

4 Data Representation Conversion

This section introduces the concept of structure metrics. They provide runtime type-information to convert complex data types and resolve pointers.

Data conversion is based on type information, and is separated into an architecture-dependent part and an application-dependent part. Architecture dependence is manifested in alignments and representations of basic data types. The layout of application defined complex data types is based on the alignment and size of basic data types, however, not on their particular data representations. Thus,

```
typedef struct FieldMetric {
  int offset,    size;
  int ucfoffset, ucfsize;
  int dim;
  union {
    BasicType btype;
    struct StructMetric *sm;
  } u;
} fieldmetric_t;

typedef struct StructMetric {
  int numfields;
  fieldmetric_t *m;
  int size,    align;
  int ucfsize, ucfalign;
} structmetric_t;
```

Figure 3: Data type of *structure metric*.

the conversion of complex data types can be separated into converting the layout of the complex data structures, and the conversion of their fields after being broken down into basic data types.

The architecture-dependent information about basic data types is used within porch to generate checkpointing and recovery code for variables of basic data type.

Definition 4 A ***type metric*** is a data structure which specifies the alignment and data representation of a basic data type with respect to a particular target architecture M_i.

The porch compiler generates a type metric for each basic data type and for each architecture M_i, based on the supplied machine formats M_i. If the type metrics of a particular target architecture M_i, requested upon compilation with porch, and the checkpoint format *UCF* differ, data representation conversion is required. In this case, conversion routines must be provided for each basic data type of machine format M_i with respect to a particular choice of the *UCF* specification.

Type metrics also facilitate the generation of application-specific data layout information, which is needed at runtime to affect conversion of complex data structures. If $M_i \leftrightarrow UCF$ conversion is required, the following data structure is generated by porch.

Definition 5 A ***structure metric*** is a data structure which specifies the layout of a complex data structure with respect to both the target architecture M_i and the *UCF* specification at runtime.

Figure 3 shows a slightly simplified version of the type definitions for the structure metrics generated by porch. A structure metric resembles the functionality and structure of an abstract syntax tree, but it provides the layout and type information about a data structure at runtime rather than during compilation. The TIL compiler [9] uses a similar approach to facilitate safety proofs of code at runtime.

The following example illustrates the information contained in a structure metric. If a C program containing data structure `struct X` is input to porch, it generates a structure metric consisting of `structmetric_t X_metric` and the array `fieldmetric_t fm`:

```
struct X {            fieldmetric_t fm[2] = {
  char c[2];            { 0, 1, 0, 1, 2, Char },
  double d;             { 4, 8, 8, 8, 1, Double }
} x;                  };
                      structmetric_t X_metric =
                        { 2, fm, 12, 4, 16, 8 };
```

This example assumes the following type metrics for a basic data type `char`. Size and alignment is 1 byte for both target architectures M_i and *UCF*. For a `double`, a size of 8 bytes is assumed in both formats M_i and *UCF*. The alignment modulus of a `double` is 4 bytes in M_i and 8 bytes in *UCF*. These different alignments lead to different layouts of `struct X` on the target architecture and in the checkpoint maintained in *UCF*.

The meaning of the individual fields of the data structures `fm` and `X_metric` can be readily deduced from Figure 3. Athough not illustrated here, the `fieldmetric_t` structure may contain a pointer to another structure metric (cf. Figure 3), allowing for arbitrary nesting of structure metrics for complex data structures.

The generation of structure metrics is based on the AST representation and is part of function $\mathcal{T}$ (†).

Transformation 2 (*Structure Metric Generation*) *The forest of ASTs is traversed to accomplish the following:*

1. *Identify all complex data structures of the input program that need to be checkpointed.*
2. *Generate a structure metric for each identified data structure according to target architecture M_i and UCF specification.*

Step 1 of this transformation handles structure declarations supplied by the operating system or libraries. If an application declares a variable based on an operating system structure, it is the programmer's responsibility to ensure that this structure is portable across different operating systems.

During checkpointing and recovery, the structure metrics are used to convert complex data structures from machine format M_i into the checkpoint format *UCF* and vice versa. The structure metrics keep the runtime functions that affect this conversion simple: just two conversion functions are required, one for each direction, which traverse the structure metrics as in an AST traversal. When visiting a basic data type, its value is accessed according to the layout information, which encodes the offset from the base of the structure and its basic type, and is converted according to the type metrics. Structure metrics are not only used for data representation conversion, but are in particular necessary for pointer resolution, described in the next section. If a pointer target is a field of a structure, the structure metric is used to compute the offset of the pointer target in the checkpoint.

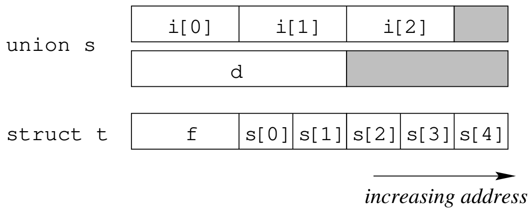
Special Case: Unions

Special consideration is necessary for checkpointing variables of complex data types which contain `unions`. Since it is impossible in general to determine at compile-time which union field is assigned before a potential checkpoint location, and would hence have to be saved in the checkpoint, additional runtime support is needed. To illustrate this problem, consider the following code fragment.

```
union {
  union {
    int    i[3];
    double d;
  } s;
  struct {
    float f;
    short s[5];
  } t;
} u;

if ( boolexpression )
  u.s.i[2] = 42;
```

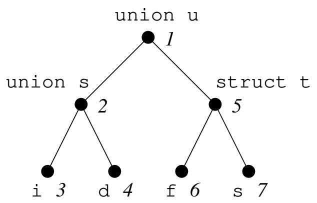
```
  else
    u.t.f = 3.14;

  checkpoint();
```

Because the value of the `if`-conditional `boolexpression` is known in general only at runtime, the decision as to whether `i`, `d` or `struct t` must be saved in the checkpoint cannot be made at compile-time. Consequently, this decision is deferred to runtime. However, compiler support is still needed, because it is also impossible in general to decide at runtime, which of the union fields is assigned if only addresses were known. A possible layout of the union in the code fragment above is:



Several variables share the same address in memory: `&u.s.i[0] = &u.s.d = &u.t.f`, `&u.s.i[1] = &u.t.s[0]`, and `&u.s.i[2] = &u.t.s[2]`. A distinction of these fields could not be made at runtime, if only an address were available.

We solve this problem by generating additional information at compile-time. Each field of a union is associated with a unique ***tag***, computed by means of a depth-first numbering of the associated AST representation of the data structure. In case of the union above, the numbered tree (array nodes can be ignored, with the exception of arrays containing unions) is:



Supplying both tag and address of the union field to the runtime system facilitates identification of the correct types for saving the data structure into the checkpoint. At runtime, the corresponding structure metric is walked in a depth-first fashion until the field associated with the tag is found. The address is necessary to distinguish different components of arrays, because values of index expressions cannot generally be determined at compile-time. The union-field information must be available during recovery as well. Consequently, an encoding of the tags is stored in the checkpoint. This encoding must be independent of the layout of the data structure to facilitate recovery on a binary incompatible machine.

In general, a nested data type may contain several unions requiring several tags to describe the field selections. In the example above, a maximum of two selections is required to distinguish whether (1) `union s` or `struct t` is stored in the checkpoint and (2), if `union s` is stored, which of its fields either `i` or `d`. To handle any nested data structure containing unions, we use an array of (union-) ***field identifiers***. Each array component corresponds to a particular selection of a union field. The array describes one or several paths in the AST. The latter case occurs if more than one field of a structure contains unions.

To supply the runtime system with the tag and address information, a new compiler transformation is introduced.

Transformation 3 (*Union-Assignment Transformation*) *All expressions with an assignment to lvalues that contain a union are transformed into a comma expression:*

expr(lvalue) → `(settag(&`*lvalue*`,` *tag*`),` *expr(lvalue)*`)`

where `settag` *is a runtime function, and tag is the unique depth-first numbering of the deepest union field in lvalue. An optimization would use data-flow analysis to determine the last assignments to a variable before potential checkpoint locations. These are the only assignments to be transformed.*

For example, the `if` statement in the code fragment above is transformed into the following code. According to the depth-first numbering, field `u.s.i` is attributed tag 3, and field `u.t.f` is attributed tag 6.

```
  if ( boolexpression )
    (settag(&u.s.i[2], 3), u.s.i[2] = 42);
  else
    (settag(&u.t.f, 6), u.t.f = 3.14);
```

The preceding presentation of the union-assignment transformation omits various details for the sake of brevity. Our current implementation limits the use of unions: Assignments via pointers to union fields cannot be handled without pointer analysis or expensive runtime checking. For example, if a pointer `int *p` were defined in the code segment above, the assignment `p = &u.s.i[2]` would make `p` a pointer to a union field. Using `p` in an *lvalue* expression such as in `(*p)++` prevents the compiler from determining the union field.

Checkpointing and recovery of unions, as well as the last assignments to unions before potential checkpoint locations are relatively expensive. Their runtime overhead comprises a depth-first traversal of the structure metric. For data structures with n AST nodes, the runtime cost is $\Theta(n)$. In addition, a space penalty affects the size of the checkpoint, because every data structure comprising unions must be supplemented with the array encoding the union tags. The size of the array is the maximum number of nested unions in the data structure.

5 Pointer Portability

This section describes how pointers become portable. Since pointers are frequently used in C programs, we discuss optimizations to reduce the space and time requirements during checkpointing and recovery.

The key idea for providing a machine-independent representation of pointers is to transform pointers into machine-independent offsets within the portable checkpoint. This transformation can only be done at runtime, because pointer targets are in general unknown at compile-time. The porch compiler generates code to support the runtime system in handling pointers.

The complexity of transforming pointers arises from the fact that the checkpoint is composed by saving objects[2] in a particular order. In fact, the checkpoint can be viewed as a stack, called ***shadow stack***, onto which objects are pushed during checkpointing, and popped during recovery. To distinguish an object's address in the active address space from that on the shadow stack, the later is called ***shadow address***. When pushing a pointer onto the shadow stack, the location of its target on the shadow stack, its target shadow address, may not be known yet. Thus, temporary storage is needed to remember that the pointer needs to be

[2] We use the term *object* to denote a variable or allocation unit. A variable defines an object through its type declaration, and a heap object is defined by the type and size information provided by its allocation call.

transformed into an offset, which can only be done when the target shadow address is known. The data structure provided by the runtime system to this end is the ***pointer stack***. Furthermore, the target shadow shadow address must be accessible. During checkpointing this address is available only when the target object is pushed onto the shadow stack. Rather than scrutinizing whether an object is a pointer target upon pushing it onto the shadow stack (as done in a previous implementation [10]), we save information about each object, including its address range, in another runtime system supplied data structure, the ***object stack***.

This setting suggests the following naive algorithm for checkpointing pointers. (1) Whenever an object is pushed onto the shadow stack, an entry containing its shadow address is pushed onto the object stack. (2) Pointers are pushed onto the shadow stack, and an additional entry containing the pointer's shadow address is pushed onto the pointer stack. (3) After all objects are pushed onto the shadow stack, all pointers on the pointer stack are resolved by looking up the target shadow address, and updating its displacement on the shadow stack in the pointer's shadow address.

In this approach, the space requirements of the pointer stack may be prohibitive. Fortunately, the number of pointer stack entries can be reduced by means of the following observation. We classify pointers based on the order in which the objects of a process are pushed onto the shadow stack.

Definition 6 A ***backward pointer*** is pushed onto the shadow stack *after* its target object.

Definition 7 A ***forward pointer*** is pushed onto the shadow stack *before* its target object.

In the naive algorithm sketched above, all pointers are treated like forward pointers. In general, the distinction between forward and backward pointers is impossible at compile time, and is therefore made by the runtime system. In the following, we describe, how this distinction can be utilized to improve the checkpointing algorithm for pointers.

Backward Pointers

Given a pointer `p` pointing to object `target`. Furthermore, assume that our checkpointing transformations generate code which saves `target` before `p`. This situation is illustrated in Figure 4. The shadow stack is assumed to grow downwards.

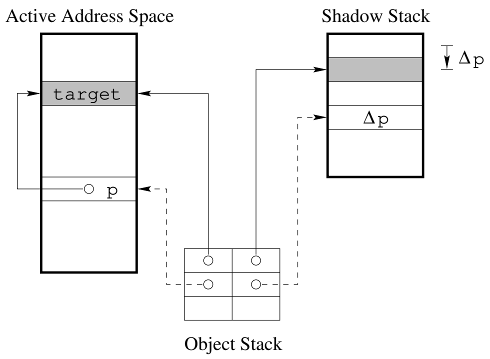


Figure 4: Checkpointing backward pointers.

Whenever an object is pushed onto the shadow stack, its address range as well as its shadow address, which is the value of the current shadow stack pointer, are also pushed onto the object stack. Before saving object `target` in Figure 4, its address range and shadow address are pushed onto the object stack (solid pointers). When visiting pointer `p`, the shadow address of `p`'s target is already saved in the target's object stack entry, which allows pointer `p` to be identified as a backward pointer: The compiler-generated code for saving pointers invokes a runtime routine to search the object stack for the target address. Since the address range occupied by `target` is already stored on the object stack, this search will be successful. Thus, the success of the search identifies pointer `p` as backward pointer.

Next, since the target shadow address is available on the object stack, pointer `p` can be transformed into the architecture independent offset Δ`p`, which is the displacement between the target shadow address and the bottom of the shadow stack. Subsequently, the offset Δ`p` is computed, and pushed onto the shadow stack.

Forward Pointers

Figure 5 illustrates the opposite situation from Figure 4: pointer `p` is pushed onto the shadow stack *before* object `target`.

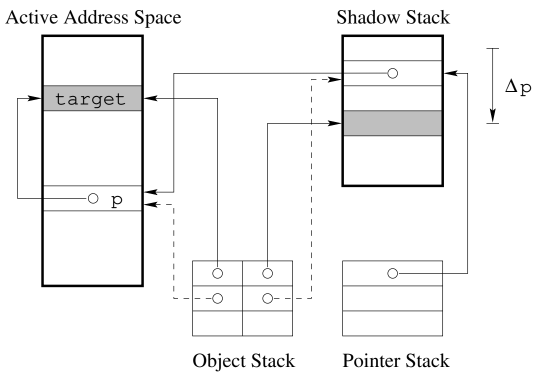


Figure 5: Checkpointing forward pointers.

Since pointer `p` will be saved before its `target`, `p` is a forward pointer. Whenever visiting a pointer, the compiler-generated checkpointing code calls a runtime routine that searches the object stack for the target object. In this case, the search will be unsuccessful, because `target` has not been pushed onto the object stack yet, which implies that pointer `p` is a forward pointer. Since the target shadow address is not known at the current state of the checkpointing procedure, the pointer needs to be saved temporarily to be resolved at a later stage. This temporary storage is provided by the pointer stack. Pointer `p` itself is pushed onto the shadow stack, and, additionally, a pointer to its shadow address, the current top of the shadow stack, is pushed onto the pointer stack.

As the checkpointing procedure proceeds, `target` is pushed onto the shadow stack. No action is taken to resolve the pointer on the pointer stack upon visiting `target`. This postponement is sensible, as it would be expensive to search the pointer stack for each object to inquire whether it is a pointer target. After all objects have been visited, a sweep through the pointer stack is necessary in order to complete the checkpoint. The information saved on the pointer stack and the object stack is sufficient to compute the offsets of all forward pointers on the pointer stack: A copy of the forward pointer itself, the target address, is stored on the shadow stack. The object stack is searched to retrieve the corresponding target shadow address. The displacement of the target shadow address with respect to the bottom of the shadow stack is computed, and stored in the pointer shadow address, still accessible from the pointer stack. This process is shown in Figure 5. A pointer to the shadow address of pointer `p` is saved on the pointer stack, which in turn holds a copy of p. This copy is used to search the object stack for the shadow address of the pointer target. The offset Δp replaces the copy of p on the shadow stack.

As demonstrated by the illustrations above, the object stack facilitates an efficient and simple way to transform pointers into machine-independent offsets at runtime. The object stack can only be employed effectively when the information provided by the structure metrics introduced in Section 4 is available. Due to the structure metrics the shadow address of a pointer target can be computed in two steps. First, the shadow base address of the target object is searched on the object stack. Second, the offset with respect to the base of the target object is converted into the shadow offset with respect to the shadow base address. It is the latter step which is based on the structure metrics. Analogously, when recovering a pointer, the structure metrics are instrumental for converting the shadow offset with respect to the shadow base address of the target object into the offset in the active address space.

Checkpointing Pointers

Algorithm 1 below describes the process of checkpointing pointers during runtime. Note, that this algorithm is integrated with Algorithm 2, which saves checkpoints and is described in Section 7, and cannot be implemented as an independent algorithm to checkpoint pointers. For the sake of clarity, however, we separate the presentation of Algorithm 1 from the remaining details.

Algorithm 1 (*Checkpointing Pointers*) *The first three steps of this algorithm are based on compiler support to identify pointers and generate calls to the runtime system.*

1. *Push an entry for the pointer or the complex data type containing the pointer(s) onto the object stack.*
2. *Before pushing the pointer onto the shadow stack, update its object stack entry with its shadow address.*
3. *Call runtime function `save_ptr` passing the pointer's address as argument. In case of complex data structures with one or more pointer fields, `save_ptr` is called for each pointer. The next two steps are executed within `save_ptr`.*
4. *Search the object stack for the target object, and retrieve its shadow address.*
5. *If the target shadow address exists, the pointer is a backward pointer, and the offset is computed and pushed onto the shadow stack. Otherwise, the pointer is a forward pointer, and its shadow address is pushed onto the pointer stack.*
6. *After all objects have been pushed onto the shadow stack, all forward pointers on the pointer stack are resolved as follows. Pop a pointer from the pointer stack, search the object stack for the target object, compute the offset by means of the target's shadow address, and store the offset in the address popped from the pointer stack.*

Algorithm 1 is slightly simplified as no distinction is made between pointer target segments, and the special handling required for function pointers and constant string pointers is omitted. Furthermore, if the target is a structure field, a lookup in the target's structure metric is required to compute the offset of the pointer target with respect to the object's shadow base address.

A special case for data representation conversion occurs for backward pointers on the stack. During the save phase, the native machine representation of the pointer according to M_i is pushed onto the shadow stack. It is popped back onto the runtime stack during the restore phase, before it can be converted into an offset. If the *UCF*-size of a pointer (offset) is smaller than the native M_i-size of a pointer, the shadow stack does not provide sufficient space to store the pointer in M_i format. In this case pointers need to be saved temporarily in another table between the save and restore phases.

Analysis and Discussion

To facilitate fast pointer resolution, porch's runtime system implements the object stack by means of three data structures, a stack, a dynamic table, and a red-black tree [2]. The stack holds entries for data/bss and runtime stack objects, the dynamic table holds entries for heap objects, and both stack and dynamic table are augmented with a red-black tree to effect fast searching. Rather than using a hash table we chose the more complex red-black tree, because it supports the search operation in a straightforward manner. The search key is an *address* that must be matched with the *address range* of an object.

Given the data structures above, the asymptotic performance of checkpointing and recovery can be derived. For a total of N_{obj} objects, pushing or popping each object onto or from the shadow stack during checkpointing and recovery respectively, requires roughly $\Theta(N_{obj})$ time (bluntly assuming that the size of an object is only a secondary contributor). Furthermore, each object is pushed onto the object stack. The dominating cost here is the insertion into the red-black tree, which requires $\Theta(\log N_{obj})$ steps. The time for pointer resolution during checkpointing, as well as for recovery is $\Theta(N_{ptr} \log N_{obj})$, because for each of the N_{ptr} pointers a target search is required over the red-black tree. Consequently, the total time for both checkpointing as well as recovery is

$$\Theta\big((N_{obj} + N_{ptr}) \log N_{obj}\big).$$

Hence, for programs with few pointers, pushing objects onto the shadow stack dominates the checkpointing overhead, whereas for programs with large numbers of pointers, the pointer resolution time dominates.

Forward pointers and backward pointers are distinguished primarily in order to minimize the size of the pointer stack, and secondarily because of time efficiency. The porch compiler sorts the shadow stack push operations for checkpointing so as to minimize the number of forward pointers. Given a lexical scope (global or local), pointerless objects are pushed onto the shadow stack first, followed by those objects containing pointers.

The lexical scope of variables can be exploited even further to reduce the bookkeeping overhead of the object stack. *Global variables* can be pushed onto the object stack right after program startup or recovery. The number of object stack insertions of a global object is therefore reduced to once per program run or recovery. *Local variables* can be pushed as soon as the frame is entered during checkpointing or recovery. Since the sequence of stack frames is in general different from checkpoint to checkpoint, a stack data structure is well suited for pushing and popping entries for runtime stack objects. Entries for local variables can be pushed and popped during checkpointing and recovery without touching the global variable entries at the bottom of the object stack. *Dynamically allocated variables* are treated slightly differently, and are accessible for checkpointing via a used-list, as explained in Section 6 below. The disadvantage of early pushing of object stack entries is that an additional red-black tree search is required to update the object stack entry with the shadow address upon pushing the object onto the shadow stack. This search does not affect the asymptotic performance of checkpointing or recovery, however.

Other practical contraints influenced the design of the pointer resolution algorithm. For example, different operating systems provide different address space layouts. Most systems place the stack segment above the data/bss segment, but some do not, such as Digital's OSF Unix, which allocates the stack beneath the data/bss segment. We are aware of at least three different orders in which data/bss, heap and stack are located in the address space. Thus, the management of the object stack must be independent of this order.

The portable offset representing a pointer within the checkpoint is the offset of the target shadow address with respect to the bottom of the target shadow segment of the pointer. This design decision allows for relocation of the shadow segments. Alternatively, we could have chosen the offset to be the displacement between the pointer shadow address and the target shadow address. This alternative has two disadvantages: (1) the shadow segments would have to be allocated at the same relative distance to each other. (2) During the restore phase, it would require an additional search of the object stack for the pointer shadow addresses.

6 Dynamic Memory Allocation

This section explains the specialized memory management required to support checkpointing and recovery.

In order to access all dynamically allocated objects when checkpointing or recovering, a customized memory allocator is provided in porch's runtime system. It provides functions included in most C libraries, currently `malloc`, `calloc`, `realloc`, and `free`. The memory allocator maintains a dynamic table of *memory block headers*, which are linked into a *free-list* or a *used-list* respectively. The used-list is implemented as red-black tree, already mentioned in the previous section. Also, the used-list is slightly modified to provide the object stack functionality for heap objects.

During allocation, a memory block header is inserted into the used-list, and possibly removed from the free-list. Freeing memory involves removing a header from the used-list, and inserting it into the free-list. Compared to most memory management algorithms, allocation and freeing involve the additional cost of inserting and deleting heap objects from the red-black tree. Nevertheless, searching a pointer target is fast when done on the red-black tree requiring only $\Theta(\log N_{obj})$ steps.

Checkpointing the heap involves scanning the used-list, and pushing all heap objects on this list onto the shadow stack. Pointers into the heap are resolved by searching the red-black tree for the target address. Maintaining a used-list allows furthermore for integrating garbage collection of the heap with the checkpointing process.

Dynamic memory management imposes two limitations on C programs: (1) The memory management provided by porch's runtime library must be used, if checkpointing of the heap is desired. (2) Type discipline is demanded for dynamically allocated memory objects. The structure metric for a heap object is derived from the type cast operator preceding its allocation call (`malloc` or `calloc`). A pointer to the structure metric is stored in the used-list to facilitate conversion of the heap object as well as pointer resolution. Consequently, the program must not violate the type information supplied by the type cast across potential checkpoint locations. This limitation is relevant for C, but would not affect strongly typed languages.

7 Checkpointing and Recovery

Given the pieces of porch's checkpoint transformations — release-and-resume instrumentation, structure metric generation, and pointer handling — this section gives the high-level algorithms for portable checkpointing and recovery.

Algorithm 2 (*Checkpointing*) *The checkpointing operation* $\mathtt{checkpoint}_i^k$ *saves the state of a computation on a machine architecture* M_i *into a portable checkpoint, according to the following algorithm:*

1. *Stop computation upon visiting an activated potential checkpoint location* $k \in \mathcal{L}$.
2. *Push* $M_i \rightarrow C_{UCF}$ *converted global variables on the shadow stack, and push corresponding entries onto the object stack.*
3. *Push* $M_i \rightarrow C_{UCF}$ *converted heap objects onto the shadow stack.*
4. *Save and restore the runtime stack, including conversion* $M_i \rightarrow C_{UCF}$ *after restoring an object.*
5. *Resolve forward pointers on pointer stack.*
6. *Store checkpoint.*
7. *Resume computation.*

Different scenarios are possible for storing the checkpoint in nonvolatile memory. For example, porch's runtime system saves the entire checkpoint to disk before resuming the computation. More clever would be to hide the communication latency by transferring the checkpoint to stable storage while continuing with useful computation [4].

Algorithm 3 (*Recovery*) *The recovery operation* $\mathtt{recover}_j^k$ *restores the state of a computation on a machine architecture* M_j *from a portable checkpoint according to the following algorithm:*

1. *Load checkpoint into shadow stack.*
2. *Pop* $C_{UCF} \rightarrow M_j$ *converted global variables from shadow stack into the data/bss segment, and push corresponding entries onto the object stack.*
3. *Pop* $C_{UCF} \rightarrow M_j$ *converted heap objects from shadow stack into the heap.*
4. *Restore runtime stack, including conversion* $C_{UCF} \rightarrow M_j$.
5. *Resolve (recovery-) forward pointers*[3] *on pointer stack.*
6. *Resume computation.*

8 Experimental Results

This section presents an empirical evaluation of the performance of checkpointing and recovery. It is not attempted to present a thorough analysis or to verify the generality of the source-to-source compilation approach. Previously published results on the performance of checkpointing and recovery have already shown that the overhead of portable checkpointing is negligible for reasonable checkpointing frequencies [10, 15]. Here, we present evidence that the performance of our new pointer resolution algorithm is acceptable as well.

Linked List

The checkpointing and recovery overhead is different for forward and backward pointers. A simple program serves as microbenchmark to study asymptotic performance. A singly-linked list of N elements with two fields, a `long` value and a `next` pointer, is allocated elementwise. In one experiment, list elements are appended to the list, which results in all pointers being forward pointers, and

[3] The definitions of backward pointers (Definition 6) and forward pointers (Definition 7) are slightly different for recovery: Substitute *active address space* for *shadow stack*.

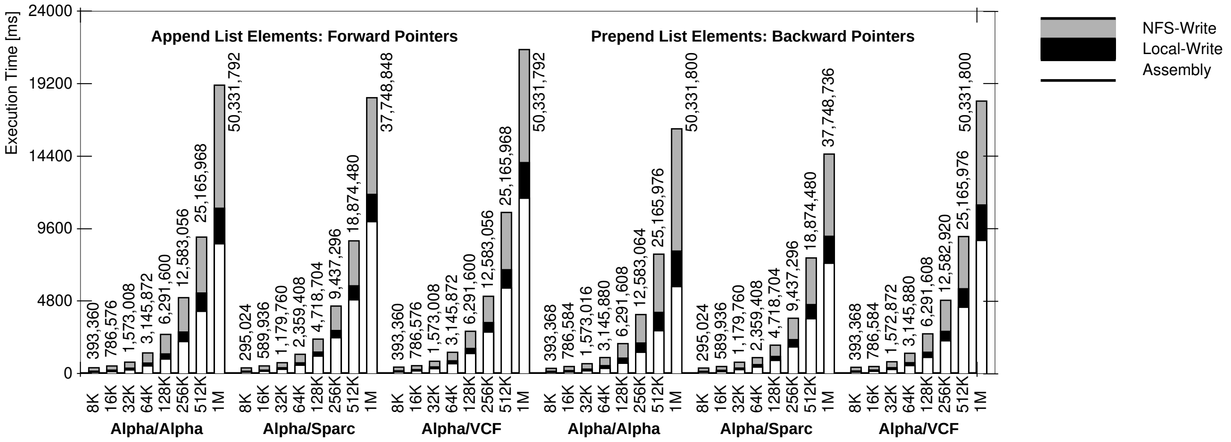


Figure 6: Checkpointing overhead for forward and backward pointers on a Digital AlphaServer 4100 for numbers of list elements between 8K and 1M. NFS-write and Local-write are overlapped, i.e. the top of both hatched area and black area give the total time for either writing the checkpoint to remote disk or to local disk. Checkpoint sizes are given in bytes on top of each bar. The pair of machine formats should be interpreted as <target architecture>/<UCF>.

in another experiment, the list elements are prepended to the list, resulting in all pointers being backward pointers.

This property is guaranteed by our dynamic memory management, which ensures that newly allocated list elements are *appended* to a dynamic table of memory block headers. During checkpointing this table is traversed such that the nth memory block allocated is also the nth to be checkpointed. No memory `free`'s are executed in the linked-list program, which could modify this order. Thus, if elements are appended, list element n is checkpointed before list element $n + 1$. Consequently, since n's `next` pointer target is element $n + 1$, n's `next` pointer is a forward pointer. If elements are prepended, element $n + 1$ of the list is allocated before element n, is checkpointed prior to element n, and its `next` pointer is therefore a backward pointer.

A single bar in Figure 6 contains the break-down of the overhead for a single checkpoint into assembly on the memory-mapped shadow stack (white portion), and the write time to either local disk (black portion) or to remote disk via NFS (hatched portion). Recovery performance is not given here, because it is almost the same as checkpointing performance.

A set of bars corresponds to a particular choice of target architecture and UCF specification, and consists of measurements for different numbers of list elements, ranging from 8,192 (8K) elements to 1,048,576 (1M). *Volker's Checkpoint Format VCF* defines 64-bit pointers and big endian byte order.

The results of this experiment can be summarized as follows. (1) The checkpoint size is of order $\Theta(N)$. (2) Checkpointing time is of order $\Theta(N \log N)$. The constants of the time bounds for checkpoint assembly across all measurements are $0.39\,\mu s$ for the lower bound and $0.86\,\mu s$ for the upper bound. (3) Checkpointing of forward pointers is 30 % to 50 % more expensive than backward pointers. (4) Data representation conversion overhead is smaller than 50 %; conversion between Alpha and Sparc involves swapping the byte order, and converting 32-bit pointers on the Sparc into 64-bit pointers on the Alpha and vice versa. Checkpoint assembly of the linked list is approximately as expensive as transferring the checkpoint via NFS to a remote file server. Flushing the checkpoint to local disk (not even a memory based file system) is comparatively cheap.

The space requirements of the pointer stack are as follows. In the case of forward pointers, it must hold $N - 1$ entries. Since a pointer stack entry consists of two pointer fields, this amounts to a pointer stack size as big as the linked list itself. No entries are necessary in case of backward pointers.

Molecular Dynamics Code

We measured the checkpointing and recovery overhead for a 2D short-range molecular dynamics code, written by Greg Johnson, Rich Brower, and the author. The data structures involved consist of linked lists of particles, neighbors, and cells.

A popular performance measure of molecular dynamics codes is the time per iteration [1]. We measured the overhead of checkpointing and recovery in number of iterations of the original code. Figure 7 presents the overhead of checkpointing and recovery. Figure 8 shows the asymptotic performance of the original code for different numbers of particles on four different machines.

Figure 7 confirms the expectation that both checkpointing and recovery overhead are roughly proportional to the number of particles. Each bar combines checkpointing and recovery overhead. The read and write times correspond to checkpoints stored on local disk. To determine the migration time of a process, the checkpoint transfer time had to be added. Figure 7 shows two sets of bars for each of the target architectures reported in Figure 8. One set corresponds to checkpoints being stored in native machine format, and for the other set, checkpoints had to be converted to and from the *VCF* format. Only the AlphaServer exhibits nonnegligible data representation conversion overhead.

No special attention has been payed to code optimization. We used `gcc -O` for native compilation on all machines. The `gcc` installation on the SGI Origin operated the machine in 32-bit mode. Furthermore, we did not gather checkpointing data on the Origin for more than 100,000 particles due to disk quota restrictions. On all other machines, checkpoint data have been collected up to the number of particles that allowed for assembling the checkpoint in memory without causing excessive swapping. The only execption is the missing data point for 500,000 particles on the Sun Enterprise Sparc-architecture with the Sparc format as UCF specification. Here, the checkpoint would be too big for 32-bit portable pointers, 3 bits of which encode the pointer target segment. The

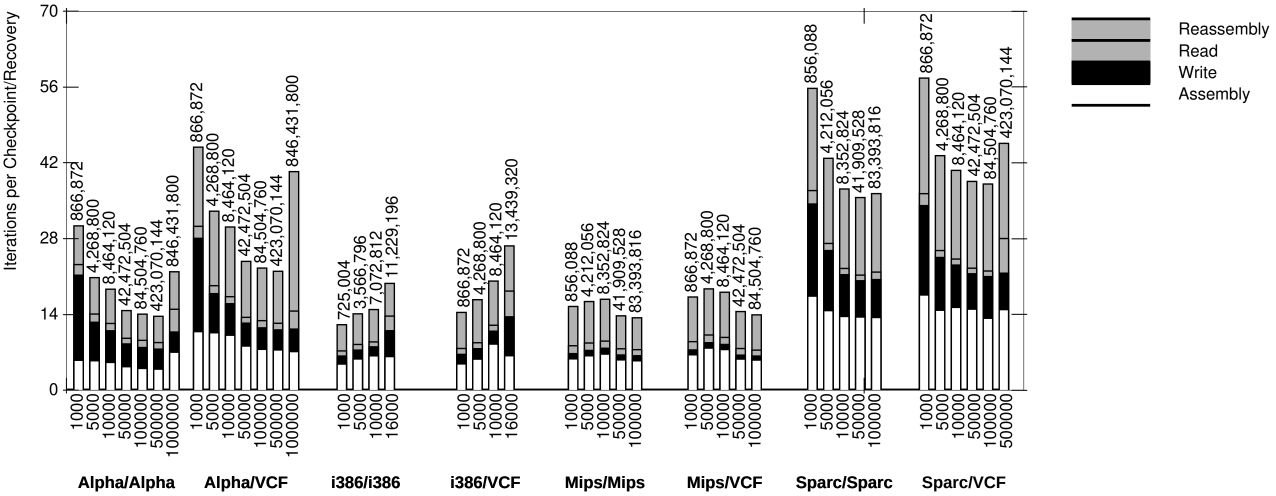


Figure 7: Checkpointing (assembly and write) and recovery (read and reassembly) overhead in number of iterations for varying numbers of particles; cf. Figure 8. Checkpoint sizes are given in bytes on top of each bar. The pair of machine formats should be interpreted as <target architecture>/<UCF>.

VCF format uses 64 bits for the portable pointer representation.

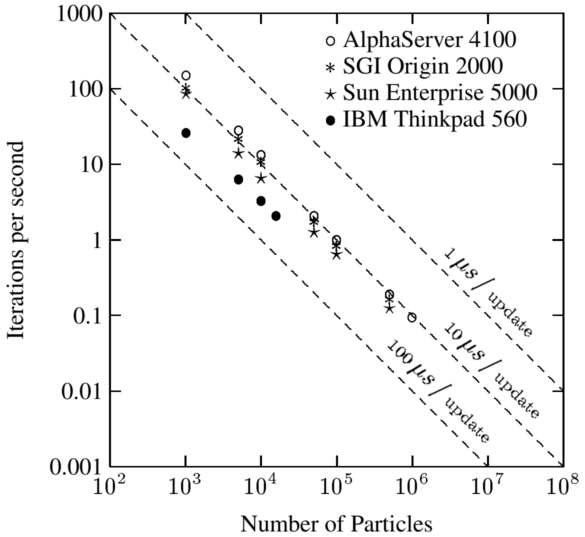


Figure 8: Performance of original 2D molecular dynamics code.

Two minor anomalies can be observed. Both the AlphaServer and the Sun Enterprise exhibit relatively low checkpointing performance for few particles as well as for high numbers of particles when memory utilization approaches the size of the available RAM. Since even in the latter case no swapping is involved, we assume the virtual memory system to be responsible.

Concluding this experiment, "reasonable" checkpointing frequencies can be chosen based on the data provided in Figure 7. For example, checkpointing every 1,000 iterations restricts the overhead to about 1 %. We regard this frequency as reasonable since the system MBTF is usually much larger, even for the largest numbers of particles that can be simulated in core memory.

It should be emphasized that the number of forward pointers depends on the order in which the linked lists are allocated. The results presented here are those for the least number of forward pointers, N for N particles. In the worst-case allocation order, more than 10 times as many forward pointers are generated. As a consequence, checkpointing performance degrades by about 40 %, and the pointer stack grows by a factor of 10. Thus, it may pay off to tune programs accordingly, in particular if space is an issue.

9 Related Work

The work presented in this article extends the ideas presented in [10, 15]. The structure metrics and pointer resolution algorithms are new and are a product of the constraints that arose during the design and implementation of porch.

Checkpointing as well as process migration are well established related areas. Good surveys on checkpointing can be found in [3, 4], and on process migration in [12]. Only little work has been done for heterogeneous environments, however. Integrating provision for portablility and fault-tolerance in a compiler tool appears to be a new approach.

The Tui system [13] is conceptually the closest work. It provides compiler support for process migration in heterogeneous systems. Tui modifies a compiler (ACK) to provide runtime information via debugging code. This approach allows Tui to handle multiple source languages easily. Tui is tied to a particular native compiler, however.

Seligman and Beguelin [11] have considered portable checkpointing and restart algorithms in the context of the Dome C++ environment. To avoid checkpointing the stack, they restrict programs to be written in form of a main loop that computes and checkpoints alternately. Furthermore, the problem of handling data structure conversion in the presence of pointers is not addressed.

Hofmeister [6] developed techniques for "dynamic reconfiguration" of software in heterogeneous distributed systems, including the transfer of software modules to another machine while an application is executing. This work implements source-to-source transformations by means of UNIX tools, such as `cxref` and `awk` rather than using compiler technology. It is not clear, how serious the limitations of this approach are. It would be straightforward, however, to implement dynamic reconfiguration with porch.

Steensgaard and Jul [14] implemented object migration within the framework of the Emerald system. They introduce *bus stops*, similar to potential checkpoint locations. For each bus stop code is generated to translate the stack environment into a machine-independent stack frame format and back. This method requires low level knowledge of the stack environment.

Theimer and Hayes [16] have presented a recompilation-based approach to heterogeneous process migration. In their scheme, a portable migration program is generated each time that a checkpoint is to be taken. This migration program is recompiled on the migration target machine and used to rebuild the process state. Our approach is conceptually simpler as we instrument the original program with code that barely affects the runtime during normal execution, and avoids the overhead of generating and recompiling a migration program.

Zhou et al. [17] describe the Mermaid system for distributed shared memory on heterogeneous systems. Mermaid uses "utility software" to generate conversion code, rather than utilizing the information provided by the abstract syntax tree to this end. Furthermore, Mermaid dedicates entire virtual memory pages for allocation of data of a particular type, which sacrifices transparency of memory allocation and is impractical for the stack.

Other applications of compiler technology for supporting checkpointing include identification of potential checkpoint locations in programs [7], and providing fault-tolerance for real-time systems [5].

10 Conclusions

This article introduces source-to-source transformations and algorithms to translate C programs into C programs capable of saving and recovering from portable checkpoints. We implemented the prototype compiler porch demonstrating the feasibility of the approach. Source-to-source compilation is the key to integrating portability and fault-tolerance, which is hard — if not impossible — to achieve with purely system-based techniques.

Using porch releaves the programmer from coping with a large number of hazards presented by heterogeneous distributed systems without compromising the low-level control offered by imperative languages like C. It is required, however, that a program compiles and runs on all target systems without modifications to the source code. Furthermore, a clean C programming style is expected, since porch computes checkpointing and recovery code based on type information.

Acknowledgements

Balkrishna Ramkumar has been co-designing porch's predecessor `c2ftc` until he decided to make a fortune outside academia. Charles Leiserson provided the stimulating environment, which enabled the evolution of porch. He also coined the name porch. The parser and type checker have been borrowed from the identity compiler `c2c` [8], written by Rob Miller, based on the Alewife C compiler from Eric Brewer and Michael Noakes. The `c2c` compiler served as a starting point for porch, which probably would not exist if `c2c` had not been available. Igor Lyubashevskiy implemented various transformations for porch, and improved the quality of the compiler by rectifying numerous "features" in the AST generation. Furthermore, thanks to Edouard Bugnion and Charles Leiserson for their feedback.

References

[1] David M. Beazley, Peter S. Lomdahl, Niels Grønbech-Jensen, Roscoe Giles, and Pablo Tamayo. Parallel Algorithms for Short-Range Molecular Dynamics. In Dietrich Stauffer, editor, *Annual Reviews of Computational Physics*, number 3, chapter 4. World Scientific Publishing Co., October 1995.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[3] Geert Deconinck, J. Vounckx, R. Cuyvers, and R. Lauwereins. Survey of Checkpointing and Rollback Techniques. Technical Report O3.1.8 and O3.1.12, ESAT-ACAA Laboratory, Katholieke Universiteit, Leuven, Belgium, June 1993.

[4] E. N. Elnozahy, David B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message Passing Systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, September 1996.

[5] A. K. Ganesh, T. J. Marlowe, A. D. Stoyenko, M. F. Younis, and J. Salinas. Architecture and Language Support for Fault-tolerance in Complex Real-Time Systems. In *2nd IEEE Conference on Engineering of Complex Computer Systems*, pages 314–322, Montreal, Canada, October 1996. IEEE.

[6] Christine R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, Computer Science Department, University of Maryland, College Park, 1993. (`file://thumper.cs.umd.edu/files/docs/3210.ps.Z`).

[7] 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.

[8] Robert C. Miller. A Type-checking Preprocessor for Cilk 2, a Multithreaded C Language. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1995.

[9] George C. Necula. Proof-Carrying Code. In *24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[10] Balkrishna Ramkumar and Volker Strumpen. Portable Checkpointing for Heterogeneous Architectures. In *Digest of Papers — 27th International Symposium on Fault-Tolerant Computing*, pages 58–67, Seattle, Washington, June 1997. IEEE Computer Society.

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

[12] Jonathan M. Smith. A Survey of Process Migration Mechanisms. *ACM Operating Systems Review*, 22(3):28–40, July 1988.

[13] Peter W. Smith. *The Possibilities and Limitations of Heterogeneous Process Migration*. PhD thesis, Department of Computer Sience, University of British Columbia, October 1997. (`http://www.cs.ubc.ca/spider/psmith/tui.html`).

[14] Bjarne Steensgaard and Eric Jul. Object and Native Code Thread Mobility Among Heterogeneous Computers. In *15th ACM Symposium on Operating System Principles*, pages 68–78, Copper Mountain Resort, CO, December 1995.

[15] Volker Strumpen and Balkrishna Ramkumar. Portable Checkpointing and Recovery in Heterogeneous Environments. Technical Report ECE-96-6-1, Departement of Electrical and Computer Engineering, University of Iowa, June 1996. (`http://www.eng.uiowa.edu/~strumpen/ece-96.6.1.ps.Z`).

[16] Marvin M. Theimer and B. Hayes. Heterogeneous Process Migration by Recompilation. In *11th International Conference on Distributed Computing Systems*, pages 18–25, Arlington, TX, May 1991. IEEE.

[17] Songian Zhou, Michael Stumm, Kai Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, September 1992.