PORTABLE AND FAULT-TOLERANT SOFTWARE SYSTEMS

PORTABLE, FAULT-TOLERANT SOFTWARE DEVELOPMENT FOR NETWORKS OF BINARY-INCOMPATIBLE MACHINES CONTINUES TO CHALLENGE ENGINEERS. PORTABLE CHECKPOINTS—SAVED AND RECOVERED ACROSS THESE MACHINES—OFFER A POTENTIAL SOLUTION.

Volker Strumpen
Yale University

High availability is essential to heterogeneous computer networks, which are the basis of many systems ranging from the Internet to fly-by-wire flight controls. Development of highly available systems, however, is constrained by ever shorter times to market and the availability of off-the-shelf hardware and software (see the "Examples" box). Consequently, the economic necessity of using commodity products from different vendors puts a premium on the products' fault tolerance. The development of fault-tolerant and portable software, particularly for parallel and distributed systems consisting of networks of binary-incompatible machines, continues to challenge engineers.

In this article, I describe a new approach to developing fault-tolerant software. This approach has been validated by a prototype compiler developed by me and my MIT colleagues as part of ongoing research. Our primary goal is to develop source-to-source compiler technology that simplifies the process of adding fault tolerance to a computation.

The programmer precompiles a program before generating an executable with a native compiler. The precompilation automatically generates code to save and recover from portable checkpoints, which capture the state of a computation in a machine-independent format. Portable checkpoints can be saved in a file or replicated on other machines in a network, and can be used to restore the computation on a binary-incompatible machine. We assume that the source program is likely to be correct, independently of whether it incorporates software reliability. Checkpointing a computation enables a restart on another machine in case of failure, regardless of whether the hardware or another software module causes the failure.

Checkpointing for fault tolerance

Well-known existing techniques to implement highly available systems, whether hardware or software, include replication with majority voting and analytic redundancy. Replication with majority voting requires a set of identical modules, and determines the set's output to be that of the majority of modules. This method is also called N-version programming when applied to provide software reliability. Analytic redundancy employs a less complex backup system to replace a high-performance module in case of failure. These methods are well suited to provide fault tolerance in a variety of settings. For computer networks, less complex and less expensive methods often suffice, however.

In computer networks, compute engines

0272-1732/98/$10.00 © 1998 IEEE

are inherently redundant, and fault tolerance can be implemented purely in software. Checkpointing is a widely used software technique. A program is supplemented with checkpointing code, which stores the computation's state in a checkpoint and recovers the computation after detecting a failure. Computations fail if their host machines fail. In a network of machines, aborted computations can be recovered on another machine.

From a computation's point of view, successful termination requires shielding itself from faults of both hardware and other software components. To date, software errors reportedly cause about 90% of computer system outage.[1] Therefore, adding fault tolerance to critical computations rather than to the underlying or surrounding software and hardware layers is justified in the spirit of an end-to-end argument.

Low-overhead checkpointing implementations are essential for this technique, because the overhead lies on the critical path and is added to the execution time of a computation. Studies have shown that the overhead of consistent checkpointing can be just a small percentage of the overall program execution time for reasonable checkpointing frequencies.[2] Our studies show that incorporating data representation conversion into the checkpointing process increases the checkpointing overhead by only 50%, roughly.[3,4] Machine-independent portable checkpoints, therefore, represent a promising mechanism for efficiently providing fault tolerance in heterogeneous environments.

Portable checkpoints

Portable checkpoints capture a computation's state in a machine-independent Universal Checkpoint Format. The UCF is a parameterizable format that specifies data representations of basic data types, such as `int` or `float` in C. The specification includes size, alignment, and byte order of all basic data types, and bit representations of floating-point numbers. The UCF can match an existing machine's format or define some virtual format. Users choose a particular UCF to trade off checkpoint size, data representation accuracy, and execution time required for data representation conversion. In a heterogeneous computer network, for example, selecting the UCF to match that of the majority of used machines is a sensible trade-off.

Most implementation techniques previously developed for checkpointing systems do not support heterogeneity. Among these are transparent methods, which require no explicit program changes other than linking a library. The Libckpt library[5] or Manetho[6] provides transparent checkpointing for homogeneous environments of binary-compatible machines. These save the values stored in the register set and the user address space into a checkpoint. Various performance optimizations, such as incremental checkpointing, rely on the virtual memory system to identify modified pages and copy-on-write to hide the latency of saving the checkpoint.

Some checkpointing systems save the state that defines the interface between the computation and the external environment, including file pointers or communication handles. Most of these systems support checkpointing only across binary-compatible machines. Furthermore, they do not support consistency of the computation and its environment, because the interface state itself is insufficient to implement operations such as transactions, as explained below.

Examples

As the time to market for new general-purpose processors shrinks, and as operating systems become a commodity, custom design of fault-tolerant systems that deliver performance competitive with that of mainstream commercial products grows unprofitable. The challenge is to build fault-tolerant systems that harness the market forces by using off-the-shelf hardware and software components, without modification. The Boeing 777, one example of a successful market-driven embedded design, contains a large embedded computer network using tens of binary-incompatible microprocessors.[1] Another market-driven project is NASA's Remote Exploration and Experimentation Project.[2] Its goal is to demonstrate the feasibility of a low-power, scalable, fault-tolerant, high-performance computing system in space using commercially available components. The NASA project's research focuses on software-implemented fault tolerance to protect presumably correct applications from frequent transient errors in the space environment.

References

1. R.J. Pehrson, "Software Development for the Boeing 777," *CrossTalk—The J. Defense Software Eng.*, Vol. 9, No. 1, Jan. 1996.
2. J.A. Rohr, "Software-Implemented Fault Tolerance for Supercomputing in Space," *Proc. 27th Int'l Symp. Fault-Tolerant Computing*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1998; Fast Abstract, http://www.chillarege.com/ftcs/fastabstracts/373.html.

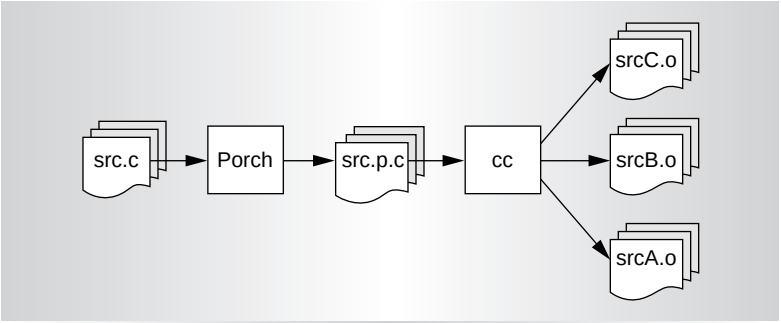

Figure 1. Source-to-source compilation with Porch. The Porch compiler precompiles a C program (src.c) to generate code for saving and recovering from portable checkpoints. Native C compilers (cc) compile the resulting program (src.p.c) to generate object codes for different target architectures.

Support for heterogeneous environments has been explored in the context of migration of computations across binary-incompatible machines. Two recent approaches are the Tui system and the work on dynamic reconfiguration.[7,8] Both systems rely on compile-time and runtime support to capture the state of a computation and convert the state into the format of a binary-incompatible machine. The methods developed for these systems are actually very similar to those for portable checkpointing. What distinguishes our approach is how it uses type information during source-to-source compilation to generate efficient code for saving and recovering from portable checkpoints.

Type information is a key to portability. It serves as an invariant for code generated for data representation conversion between binary-incompatible machines. A second key is the checkpointing code generated at the programming-language level. This allows for generating checkpointing code that accesses variables by their names and makes the checkpointing code independent of low-level details, such as register allocation and stack environment.

It is not clear whether it's possible in general to convert the state of a computation from one system to another using conversion code at the assembly level. Different systems have different register set designs, hidden registers, hardware support for function calls, stack layouts, and so on. Consequently, system-based approaches such as transparent checkpointing are generally not suited for transforming the state of a computation into a machine-independent format. Portable checkpointing, however, presents a potential solution.

The Porch compiler

We designed the Porch compiler to explore the use of automatic code generation for portable checkpointing.[3,4] The Porch compiler is a source-to-source compiler that translates C programs into semantically equivalent C programs additionally capable of saving and recovering from portable checkpoints. We chose the C programming language for our prototype because it is widely used.

To enable code generation for portable checkpointing and recovery, we identify potential checkpoint locations in a C program by inserting a call to the library function `checkpoint()`. These locations are hooks for program analysis and code generation. Unlike other approaches that save a checkpoint when receiving an asynchronous signal, computations precompiled with Porch save a checkpoint when visiting the next potential checkpoint location after a checkpoint request is received. Potential checkpoint locations can be inserted automatically or by a programmer. In contrast to automatic insertion, a programmer will generally identify well-suited checkpointing locations in a program, which results in lower execution-time overhead.

Figure 1 illustrates Porch used as a precompiler. The user precompiles a C program with Porch, then uses native C compilers to generate the object code for different systems. The Porch compiler parses the input source, generates an abstract-syntax-tree representation of the input, and performs type checking. The Porch compiler therefore has access to the complete source code and to its type information.

Programmers might view Porch as a quality-assurance tool. The code analysis performed by Porch at compile time reveals code fragments that may lead to inconsistent checkpoints. Furthermore, the checkpointing code generated by Porch inspects the entire live state of a program, when it saves a checkpoint at runtime. The Porch compiler punishes negligent programming style (such as dangling pointers) by refusing to save a checkpoint and delivering an error message instead. Potential checkpoint locations in a program are thus true checkpoints for a thorough programming style in unsafe languages such as C.

Automatic code generation for saving and recovering portable checkpoints consists of

two conceptually separate, although tightly integrated, stages. During the first stage, Porch generates code for saving and recovering a computation's internal state. During the second stage, Porch generates code for saving and recovering the computation's environment. The second stage preserves consistency between the internal state and the external state of a computation.

Stage I: Code generation for internal state

The internal state of a computation consists of the live variables that Porch identifies. Checkpointing of this state resembles garbage collection. The computation stops, and the checkpointing code generated by Porch copies its internal state into a checkpoint. The Porch compiler views the checkpoint as a stack. Porch generates stack operations to push the values of the internal state onto the stack during checkpointing and to pop them from the stack during recovery. Both push and pop operations access variables by their name, with the exception of dynamically allocated variables, discussed later.

The checkpointing code generated by Porch converts data representations on the fly while pushing and popping variables onto the checkpoint. The Porch compiler automatically generates code for basic data type conversion, such as little endian to big endian or vice versa. The Porch compiler effects conversion of structures and unions (complex data types) by means of a so-called structure metric. Rather than generating conversion functions for each structure or union defined in the input program, Porch generates structure metrics for all declarations of complex data types. Generic runtime functions are invoked during checkpointing and recovery to traverse the structure metric and to convert data representations.

Figure 2 illustrates a simplified example of a structure metric in C. We assume that `struct X` has different layouts on the target architecture and in the checkpoint. The difference in the example stems from the alignments of the `double` type, assumed to be a 4-byte boundary on the target architecture in Figure 2b and an 8-byte boundary in the checkpoint in Figure 2c. Thus, two padding bytes separate the `char` array and the double on the target architecture, whereas six bytes pad the structure in the checkpoint.

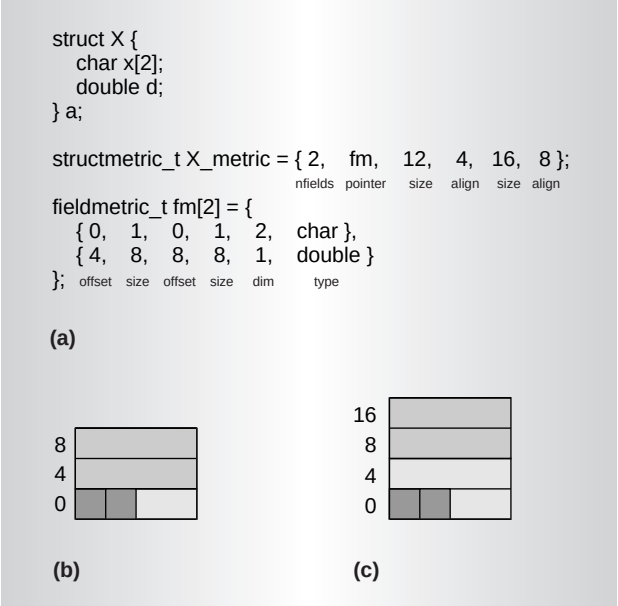

Figure 2. The compiler-generated structure metric (a) specifies the layout of complex data types for both the target architecture (b) and the checkpoint format (c) at runtime. During checkpointing and recovery, the automatically generated code converts data representations and structure layouts by traversing their associated structure metric.

The structure metric describes size and alignments of `struct X` and contains a pointer `fm` to an array of field metrics describing each individual field. The structure metric contains two pairs of size and alignment values: one pair for the structure on the target architecture and one for the structure in the checkpoint layout. Similarly, each field-metric component contains two pairs of offset and size values: one for the target architecture and one for the checkpoint layout. The offset denotes the field offset from the structure's base. The dimension denotes the total dimension of a potentially multidimensional array. The type field may also hold a pointer to another structure metric, allowing for arbitrary nesting of complex data types.

Saving the state of a computation on a stack and introducing structure metrics to effect data representation conversion is not sufficient to implement checkpointing and recovery for most programming languages. Certain C language features require other techniques, three of which I discuss briefly: pointers, the stack environment, and dynamically allocated variables on the heap.

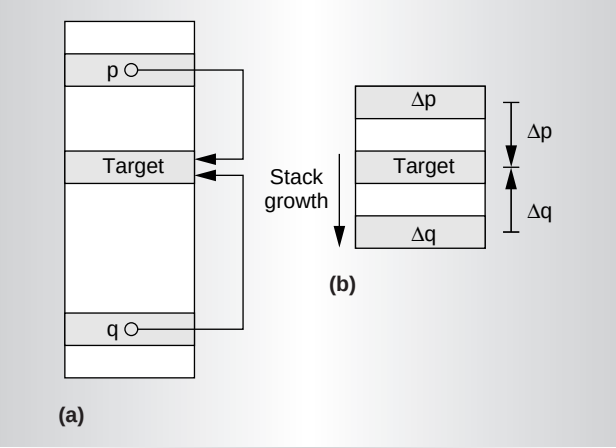

Figure 3. The checkpointing code generated by Porch transforms pointers from the address space (a) into machine-independent offsets in the checkpoint (b). The offset computation requires runtime bookkeeping, because the checkpointing code may not have the address of a pointer target in the checkpoint when pushing the pointer itself.

Checkpointing pointers

The checkpointing code generated by Porch renders pointers portable by translating them into machine-independent offsets within the portable checkpoint. Since a pointer's target address is generally unknown at compile time, Porch generates code that enables its runtime system to perform the pointer translation during checkpointing and recovery.

Figure 3 illustrates the problem encountered when translating pointers into offsets within the checkpoint. The checkpointing code will push either a pointer or its target first onto the checkpoint stack, because pushing variables is a totally ordered sequential process. Figure 3a shows two pointers `p` and `q` with the same `target` in an address space. We assume that the address space is traversed top down, so that the order of pushing variables onto the checkpoint is `p` before `target` before `q`.

When the checkpointing code generated by Porch pushes pointer `p`, its target is not on the checkpoint stack yet and the desired offset Δp cannot be computed. When pushing `target`, the checkpointing code could compute Δp but would need further checking to discover if a variable is in fact a pointer target. Finally, when pushing `q`, the code could compute Δq if it has the `target` address in the checkpoint. This is not the case, however, unless the checkpointing code performs additional bookkeeping.

To facilitate the offset computation for pointers, Porch's runtime system maintains bookkeeping information during runtime. Figure 4 shows the bookkeeping data structures used by Porch's runtime system. For each locally or globally declared variable, the runtime system maintains an object table entry, and for each dynamically allocated object, an entry in the used list. The runtime system updates the entry associated with an object during checkpointing and recovery. The entry contains pointers to an object's address in the address space and its copy in the checkpoint. The runtime system integrates the object table and the used list with a red-black tree[9] (an extended-binary-tree data structure) to facilitate fast inserts, deletes, and searches.

With bookkeeping established, Porch's runtime system can resolve pointers by a second traversal through the checkpoint after it has been filled with the computation's live state. During the first traversal, the runtime system pushes the live state onto the checkpoint stack. Pointers are merely copied into the checkpoint. During the second traversal, each pointer is visited again. The runtime system searches the pointer's target object in the red-black tree and retrieves the target's address in the checkpoint from the object table entry. The address is used to compute the machine-independent offset within the checkpoint. The runtime system then replaces the pointer copy in the checkpoint with the offset.

The structure metric plays a central role during the offset computation. Since the layout of complex data types may differ on the target architecture and in the checkpoint, Porch's runtime system must compute the pointer offsets to structure fields accordingly. For example, a pointer to `double d` in Figure 2 has a 4-byte offset from the base of `struct X`. It has an 8-byte offset in the checkpoint. For the offset computation, Porch's runtime system retrieves the checkpoint format's offset from the structure metric. Similarly, during recovery, the runtime system retrieves the offset on the target machine from the structure metric.

Nonportable pointers—for example, to hardware-specific addresses—cannot be checkpointed in this scheme. Such pointers are not part of the portable portion of a program, however. Generally, Porch generates

code for programs that compile and run on the chosen target architectures without modifying the source code. This approach does not prevent system-specific coding such as conditionally compiled or hardware-specific code fragments, but it requires structuring a program appropriately. Programmers can hide system-specific details from Porch by structuring the program into multiple translation units and functions. The Porch compiler employs interprocedural analysis to instrument only those functions of a program on the call path to a potential checkpoint location. Those functions not on the path may safely be system-specific.

A reasonable programming style would group all system-specific functions and data into a translation unit separate from the portable code. Programmers would precompile only the portable code portion with Porch. The nonportable portion could be different for different systems, because it would not affect the state of a computation during checkpointing.

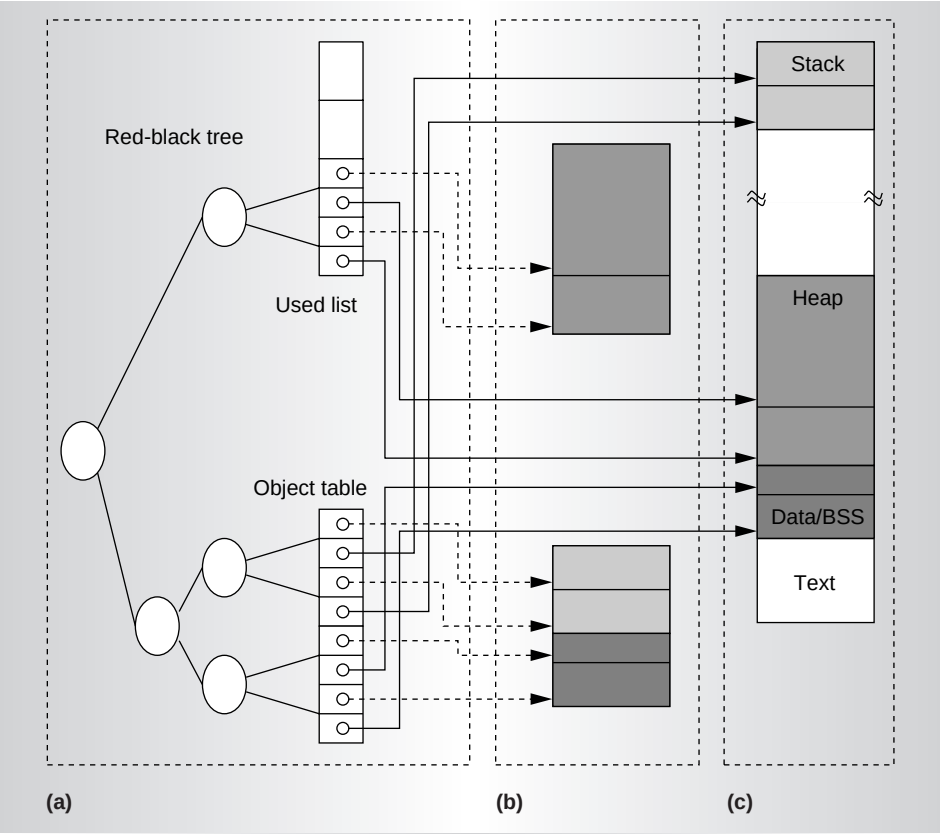

Figure 4. Internal state after checkpoint assembly. The Porch runtime system pushes the original address space onto the checkpoint. The bookkeeping data structure (a) integrates a red-black tree, the used list, and the object table. The dashed arrows point to the object's address in the checkpoint (b), and the solid arrows point to the object's address in the address space (c).

A more detailed description of pointer handling, including optimizations, can be found elsewhere.[4] Opportunities for future research include minimizing the space overhead resulting from the red-black tree implementation. Other garbage collection techniques could be used to substitute the bookkeeping data structures shown in Figure 4.

Checkpointing the stack environment

The runtime-stack environment is deeply embedded in a system, formed by hardware support, operating system, and programming language design. A key design decision was to implement Porch as a source-to-source compiler. We did so to avoid coping with system-specific state such as register allocation and runtime-stack layout, or hidden state such as the program counter.

The challenge in checkpointing the runtime stack is gaining access to local variables of active functions on the stack. By definition, lexical scoping enforces entering the lexical scope of a function to access its local variables by their names. The lexical scope of a function can be entered by redirecting a computation's control flow. The only portable mechanism for visiting function frames on the runtime stack is the standard function call-and-return mechanism.

The Porch compiler transforms a C program such that during execution, functions can be *released* from automatically generated exit points and can be *resumed* via new entry points.[3,4] When entering a function frame, the Porch-generated checkpointing code pushes the local variables onto the checkpoint. Within the lexical scope of the function, all local

The Porch compiler generates entry and exit points as well as the code for control-flow redirection with the standard function call-and-return mechanism, rather than modifying system-specific hidden state including the stack or frame pointer.

variables can be accessed by their names, which is independent of system-specific register allocation. The Porch compiler generates entry and exit points as well as the code for control-flow redirection with the standard function call-and-return mechanism, rather than modifying system-specific hidden state including the stack or frame pointer.

Figure 4 shows that the state of the runtime stack resides in the checkpoint on top of the global variables of the data/BSS segments of a typical Unix address space. This illustration assumes that the checkpoint stack grows upward. Note that the layout of the checkpoint portion containing global variables remains invariant across different checkpoints. Consequently, Porch's runtime system allocates the red-black tree portion for global variables only once when it saves the first checkpoint or during recovery. The values of global variables may change between checkpoints, however, and must be copied during checkpointing. In contrast, the active state on the runtime stack may change from checkpoint to checkpoint.

Checkpointing the heap

Checkpointing dynamically allocated variables on the heap requires type and size information to effect data representation conversion. In contrast to local and global variables, the size of heap objects is generally known only at runtime. Thus, for each heap object, type and size information is maintained at runtime. The Porch compiler generates code to update this runtime information upon object allocation.

To facilitate checkpointing, Porch's runtime system traverses the entire heap. The runtime system maintains a used list for heap objects that contains one entry for each heap object. The entry contains type and size information as well as the object's address on the heap. The used list serves as an object table when resolving pointers and is integrated with the red-black tree. Checkpointing a computation includes garbage collection on the heap. Figure 4 shows the used list integrated into the bookkeeping part of the computation's internal state.

A disadvantage of our current implementation of dynamic memory management in Porch is that it cannot be easily replaced with application-specific memory management. In principle, replacement is possible, however, because the used list data structure is orthogonal to the free list used in most memory management implementations.

Stage II: Code generation for interface state

The second stage in generating code for portable checkpointing involves the interface between the internal and external state, such as files.

The Porch compiler identifies the internal state of a computation. During checkpointing and recovery, the internal state is accessible by Porch's runtime system. In contrast, the environment of a computation contains external state that may be inaccessible by both the Porch compiler and its runtime system. Examples for external state are file contents, mouse position, and display pixels. External state may be partly visible to the computation, however. The *cognizant state* is the part accessible during checkpointing and recovery, and is the intersection of the internal state and external state. If the cognizant state does not cover the entire external state, inconsistencies between the internal state and external state may occur due to a failure.

For example, the coordinates describing the mouse position may be part of a computation's cognizant state. Recovering the mouse position from a checkpoint may be considered an inconsistency by some applications, because the actual position may have been different when a failure occurred. For other applications, it may be acceptable to recover with the mouse position that the operating system initializes. Alternatively, it may be sufficient to initialize the mouse coordinates to the center of the screen. Defining the recovery position as part of the

internal state provides sufficient information to restore a desired consistent state. Another possibility would be to update the mouse coordinates atomically as part of the cognizant state, adding more runtime overhead in favor of a higher degree of consistency.

A computation's view of its environment depends on the specification of the cognizant state. If the cognizant state is consistent with the internal and the external state, the computation's view of the environment is consistent. To reconstruct a consistent view after an inconsistency occurs, more than just the cognizant state may be needed. The notion of interface state captures the information necessary to reconstruct a consistent view of the computation's environment. The amount of interface information needed depends on the definition of consistency. Generally, the higher the degree of consistency, the more interface information is needed.

For example, consider the external state saved in persistent files. Usually, internal state and file contents should both be consistent. In most C programs, the file pointer constitutes the only cognizant state of the file, with no information kept about file operations. Figure 5 shows how an inconsistency can occur. As the computation proceeds in time, a checkpoint of the internal state is saved, and a read and a write operation are performed, then a crash occurs. After recovering the computation from the last checkpoint, file pointer X is restored, but the value read from this file location is not the expected value 42, assuming that the file is persistent. The computation thus sees an inconsistency between the internal state and the external state.

As part of our research, we have experimented with the inconsistency problem in the context of the file system. The ftIO system is a prototype that resolves inconsistencies with the external state stored in a file.[10] The ftIO system shows how interface state can be defined to support consistent checkpointing of computations that access files. The interface state maintained by the ftIO system includes the file pointer, a newly defined protocol state for the file, and file attributes including opening mode and buffering policy. Note that only the file pointer constitutes the cognizant state in a typical C program.

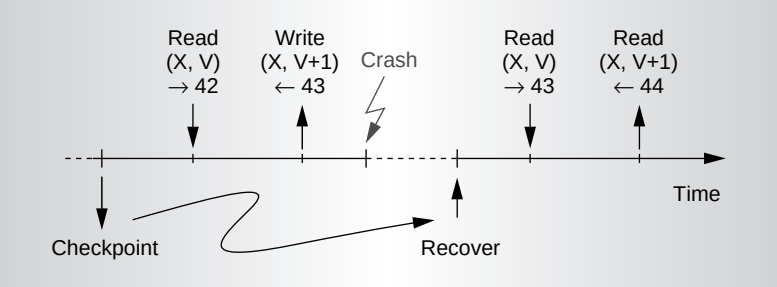

Figure 5. A failure causes an inconsistency between a computation's internal state and the external state of a file. The computation progresses on the time line. The vertical arrows denote I/O. Before the crash, value 42 is read from file position X into variable V, and value 43 is written back into location X. Since the file is persistent, the value read from position X after recovery is 43 rather than the expected value 42.

The ftIO system solves part of the broader problem of providing portable and fault-tolerant file I/O. It provides transactional file operations for fault tolerance. The Porch compiler generates code for checkpointing the interface state in a portable checkpoint by precompiling the ftIO system itself with Porch. This approach lets us implement ftIO's runtime system as a shallow layer of wrapper routines for all file operations defined in the ANSI C standard. For details, see "The ftIO system" box on the next page.

The Porch compiler technology explores source-to-source compilation to simplify the task of designing portable, fault-tolerant software systems by means of automatic code generation. Thus far, we understand how to generate code for portable checkpoints to recover a computation on a binary-incompatible machine.

Source-to-source compilation has surprisingly few limitations, even for non-type-safe programming languages such as C. If a C program is written cleanly, Porch can generate checkpointing and recovery code. Otherwise, Porch either does not generate code in case of ambiguities, or emits an error message at runtime rather than checkpointing ambiguous state.

Among the limitations for C programs is the required type-conforming use of dynamically allocated objects. Pointers may not be assigned to nonpointer types across potential checkpoint locations. Assignments via point-

The ftIO system

The ftIO system is a private-copy/copy-on-write design that copies the entire file on the first write operation and performs subsequent file operations on the replica.[1] During checkpointing, the ftIO system commits modifications by simply replacing the original file with its replica. The ftIO algorithm is based on a finite automaton built on the set of interface states for each file accessed by the application. Interface-state transitions occur when certain file operations are executed.

ftIO interface state

Each file is associated with an interface state stored in three orthogonal flags. File types can be

- *Clean or dirty:* A file is clean if it has not been modified since the last checkpoint. Otherwise, it is dirty. In our private-copy implementation, no replicas exist for clean files.
- *Open or closed:* A file can be either open or closed. Files that do not exist are closed by definition.
- *Live or dead:* A file can be scheduled for removal, in which case it is dead. Otherwise, it is alive.

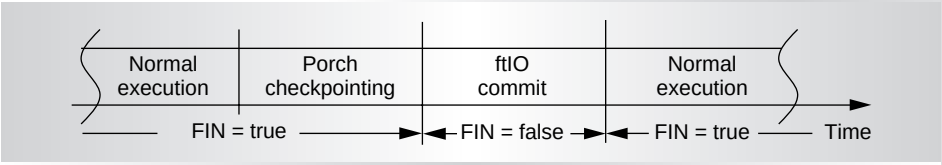

Figure A. Checkpointing phases.

Besides the file-specific states, a global Boolean ftIO state is maintained. FIN (for finished) is a bit used to record the success of the ftIO commit operation.

We distinguish ANSI C file operations from ftIO file operations. The latter define the input alphabet of the ftIO finite automaton. Most of them have equivalent C file operations. For example, the ftIO operation `create` creates and opens a file. This ftIO file operation corresponds to the C file operation `fopen` with opening mode "`w`," or "`a`" if the file does not exist already.

All ftIO file operations are implemented as wrappers around the standard ANSI C file operations. These wrapper functions maintain for each file a data structure that contains its ftIO interface state. Transitions between interface states are determined by the ftIO finite automaton.[1] The Porch compiler replaces ANSI C file operations with their equivalent ftIO file operations.

Checkpointing and recovery

We separate checkpointing and recovery into two phases. Figure A shows the sequence of the checkpointing phases between phases of normal execution. First, the Porch runtime system saves the internal state (Porch checkpointing). Second, the ftIO system commits files during the ftIO commit phase. The two phases are accompanied by a third action, the maintenance of the FIN bit.

Figure B shows the recovery cases before returning to normal execution. First, the Porch-generated code recovers the internal state (Porch recovery in Figures B1 and B2). Second, the ftIO system resolves potential inconsistencies depending on the state of the FIN bit (ftIO recovery in Figures B1 and B2).

Upon checkpointing, the Porch-generated code saves the internal state of a process in a temporary checkpoint file. The internal state includes the interface state. The runtime system replaces the previous checkpoint file by means of an atomic `rename` operation. Then, the ftIO system performs commit operations for all files. For each dirty file, the original file is replaced with its replica, and the file state transitions to clean. If a file is dead or closed, however, both the actual file and the replica (if one exists) are removed irrespective of the clean/dirty state.

The FIN bit ensures the atomicity of the ftIO commit operation. It occupies a bit in the checkpoint. The

ers to union components rather than pointers to a union itself are not analyzable by Porch. The Porch compiler does not support conversion of bit fields because they are inherently nonportable. Furthermore, programmers may need to restructure expressions with potential side effects to enable safe compiler transformations. The Porch compiler issues a warning in cases where it cannot determine if a transformation is safe. None of these limi-

runtime system initializes a new temporary checkpoint file with the FIN bit set to false. At the end of the ftIO commit phase, the runtime system sets the FIN bit to true. This happens in the actual checkpoint file, newly created during the Porch checkpointing phase. In Figure A, the FIN bit represents a global state, defined as the FIN bit value in the last checkpoint file. Therefore, the FIN bit becomes false in Figure A only after the Porch checkpointing phase, when the new checkpoint file with the FIN bit set to false is committed by replacing the previous checkpoint.

Failures may occur during normal execution, Porch checkpointing, or ftIO commit. There are only two distinguished failure cases, however, which simplifies reasoning about the correctness of the ftIO algorithm.[1]

Recovery is also split into two phases:

1. During Porch recovery, the runtime system restores the internal state of a computation and reads the FIN bit from the checkpoint file, which determines the mode for ftIO recovery.
2. The ftIO runtime system recovers files during the ftIO recovery phase.

The FIN bit serves to distinguish the two recovery modes. Figure B shows the recovery phases for both possible values of the FIN bit. If the FIN bit is true, the checkpointing process succeeded, and the ftIO recovery phase is just a `no-op`. If a failure occurs during the ftIO commit phase, the FIN bit in the checkpoint remains false. In this case, files are committed during the ftIO recovery phase by executing the commit phase again.

No special treatment is required to handle a failure that occurs during recovery. Since the ftIO recovery phase consists of executing either the idempotent ftIO commit operations or a `no-op`, the ftIO runtime system can safely execute the recovery phase again to recover from a failure.

The ftIO system illustrates the handling of an external state within the framework of the Porch compiler technology. We believe that the entire environment of a computation can be handled in this fashion. Once the interface state is identified, the routines effecting interface-state transitions are written as ordinary C programs. Compiling these routines with Porch guarantees that the interface state is checkpointed in a portable manner.

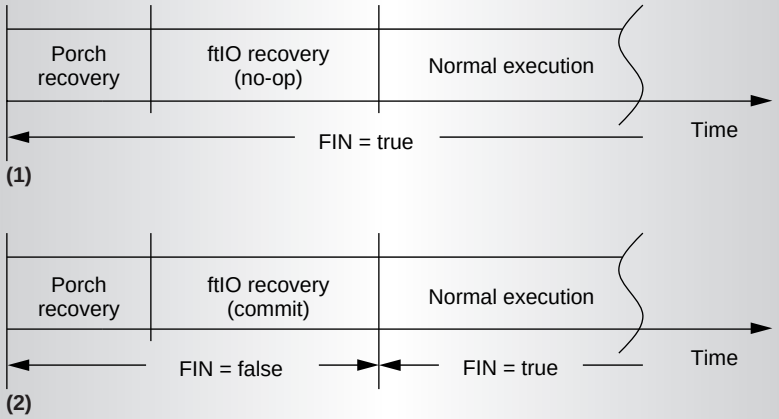

Figure B. Recovery phases if a failure occurred during normal execution or during Porch checkpointing (1), and if a failure occurred during the ftIO commit phase (2).

Providing support for transactional file operations has previously been considered. More recent work includes the `libfcp` library.[2] This work implements the undo-log concept, using a two-phase commit protocol to commit a transaction. Besides requiring explicit annotation of a program to mark the start and end of a transaction, `libfcp` does not support portability. In combination with `libckp` and `libft`, `libfcp` can be used to checkpoint the state of persistent storage in the context of checkpointed applications in homogeneous environments.[2]

References

1. I. Lyubashevskiy and V. Strumpen, "Compiler-Supported Portable, Fault-Tolerant File-I/O," *Proc. IEEE Int'l Workshop on Embedded Fault-Tolerant Systems*, 1998, pp. 49–56; http://theory.lcs.mit.edu/~strumpen/efts98.ps.gz.
2. Y-M. Wang et al., "Integrating Checkpointing with Transaction Processing," *Digest of Papers, 27th Int'l Symp. Fault-Tolerant Computing*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1997, pp. 304–308.

tations are an issue for cleaner programming languages such as Java, for example.

Our work demonstrates how compiler support enables checkpointing of a computation's environment. We believe that implementing other functionalities, including a fault-tolerant socket interface for interprocessor communication, can be achieved similarly. Conceivably, an open compiler infrastructure can be created that lets programmers specify the interface

state. Moreover, such an infrastructure would let programmers use the compiler to generate code for checkpointing and recovering the state of a particular device consistently. MICRO

Acknowledgments

This work has been funded in part by the US Defense Advanced Research Projects Agency's grant F30602-97-1-0270. Thanks to Charles Leiserson for his support. Igor Lyubashevskiy implemented the ftIO system as part of his MEng thesis. Thanks also to Matteo Frigo for stimulating discussions and for his comments.

References

1. A. Wood, "Predicting Client-Server Availability," *Computer*, Vol. 28, No. 4, Apr. 1995, pp. 41–48.
2. E.N. Elnozahy et al., "The Performance of Consistent Checkpointing," *Proc. 11th Symp. Reliable Distributed Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1992, pp. 39–47.
3. B. Ramkumar and V. Strumpen, "Portable Checkpointing for Heterogeneous Architectures," *Digest of Papers—27th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 58–67.
4. V. Strumpen, "Compiler Technology for Portable Checkpoints," to be published, 1998; http://theory.lca.mit.edu/~strumpen/porch.ps.gz.
5. J.S. Plank, M. Beck, and G. Kingsley, "Libckpt: Transparent Checkpointing Under Unix," *Proc. Usenix Winter 1995 Tech. Conf.*, Usenix, Berkeley, Calif., pp. 213–233.
6. E.N. Elnozahy, *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication*, doctoral dissertation, Rice Univ., Dept. Computer Sci., Houston, Texas, 1993.
7. P.W. Smith, *The Possibilities and Limitations of Heterogeneous Process Migration*, doctoral dissertation, Univ. of British Columbia, Dept. Computer Sci., Vancouver, Canada, 1997; http://www.cs.ubc.ca/spider/psmith/tui.html.
8. C. Hofmeister, *Dynamic Reconfiguration*, doctoral dissertation, Univ. of Maryland, Computer Sci. Dept., College Park, Md., 1993.
9. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1990.
10. I. Lyubashevskiy and V. Strumpen, "Compiler-Supported Portable, Fault-Tolerant File-I/O," *Proc. IEEE Int'l Workshop on Embedded Fault-Tolerant Systems*, 1998, pp. 49–56; http://theory.lcs.mit.edu/~strumpen/efts98.ps.gz.

Volker Strumpen is an assistant professor of electrical engineering and computer science at Yale University. Previously, he was affiliated with Sony in Atsugi, Japan, and the Massachusetts Institute of Technology. His current research interests are the design of fault-tolerant, distributed computer systems and associated programming methodology. Strumpen received a Diploma in electrical engineering from RWTH Aachen in Germany and a PhD in computer science from ETH Zurich, Switzerland.

Contact Volker Strumpen regarding this article at Yale University, Dept. of Computer Sci., 51 Prospect St., AKW-203, New Haven, CT 06520; strumpen@ee.yale.edu; http://ee.yale.edu/~strumpen.