Compiler-Supported Portable, Fault-Tolerant File-I/O

Igor Lyubashevskiy and Volker Strumpen
Laboratory for Computer Science
Massachusetts Institute of Technology
{igorlord,strumpen}@supertech.lcs.mit.edu

***Abstract*—The *ftIO*-system provides portable and fault-tolerant file-I/O by enhancing the functionality of the ANSI C file system without changing its application programmer interface and without depending on system-specific implementations of the standard file operations. The *ftIO*-system is an extension of the porch compiler and its runtime system. The porch compiler automatically generates code to save bookkeeping information about *ftIO*'s transactional file operations in portable checkpoints. These portable checkpoints can be recovered on a binary incompatible architecture. We developed a new algorithm for supporting transactional file operations in *ftIO*. Rather than using the well-known two-phase commit protocol, this algorithm uses only a single bit of information and an atomic `rename` file operation to guarantee fault tolerance. In this paper, we describe our new *ftIO* algorithm, discuss design choices for *ftIO*, and provide experimental data of our *ftIO* prototype.**

I. Introduction

High-level programming languages such as C are commonly used for programming embedded systems. This fact provides an opportunity in embedded systems design to leverage software technology that was originally developed for general-purpose systems. The porch compiler [1], [2], for example, has been developed to transform C programs into semantically equivalent C programs that are additionally capable of saving and recovering from portable checkpoints. This technology can be exploited to provide fault-tolerance in heterogeneous systems and may also be used for debugging or retrospective diagnostics [3]. To the latter end, a portable checkpoint can be used to restart or inspect a program on a binary incompatible system architecture.

In this paper, we explore extensions to the porch compiler and its runtime system to support portable, fault-tolerant file-I/O. Our studies are based on a prototype implementation, the so-called *ftIO*-system. The *ftIO*-system enhances the functionality of the ANSI C file system [4] without changing its application programmer interface and without depending on the system-specific implementation of file operations. As a result, application programs using formatted file-I/O can be made fault-tolerant by precompiling them with porch. Thereafter, object code is generated with a conventional C compiler. System-specific C-library implementations can be used without modifications, because porch generates code based on the standard ANSI C function prototypes.

The problem we are tackling can be decomposed into three subproblems: (1) providing transactional file operations for fault-tolerance, (2) checkpointing the state maintained by transactional file operations in a portable manner, allowing for recovery on a binary incompatible machine, (3) maintaining the data stored in a file in a machine-independent format. When all three functionalities are present, we can save a checkpoint of a process, including the files it accesses, and recover the computation on a binary incompatible machine. The *ftIO* system provides the first two functionalities. Maintaining file data in a machine-independent fashion can be implemented by extending the porch compiler.

We consider an *atomic transaction* to be the execution of a program between two consecutive checkpoints. The program either *commits* its state during checkpointing or *aborts* at some point during execution, in which case it can be recovered from the last checkpoint. Problems arise if a program *writes* to a file and aborts before the write has been committed. The reason is that files are (generally) kept in nonvolatile storage. A value written before a crash is likely to remain in the file and will therefore erroneously be visible after recovery.

To avoid such incorrect behavior, we designed and integrated a new protocol with the porch compiler technology. This protocol is based on the *copy-on-write* concept [5], where the entire file is copied upon the first write operation. Subsequent file operations are performed on the replica. During checkpointing, the modifications are committed by simply replacing the original file with its replica. The protocol is based on a single bit of information to ensure fault tolerance.

The *ftIO* system maintains bookkeeping information about the copy and original file to preserve consistency with the remaining state of the application. This book-

keeping information comprises among others the protocol state of the file, e.g. clean or dirty, the file position, as well as attributes such as opening mode or buffering policy. It is important to checkpoint this information in a portable manner to facilitate recovery of an application on a binary incompatible machine. We utilize the porch compiler to automatically generate code for checkpointing the bookkeeping data in a machine-independent format by precompiling the *ftIO* runtime system itself with porch. This approach allows us to implement the runtime portion of *ftIO* as a shallow layer of wrapper routines for the complete set of file operations defined in the ANSI C standard.

The porch compiler and the *ftIO* system have been developed for the design of fault-tolerant systems based on the notion of portable checkpoints. This technology may also be valuable for the design of embedded systems itself. We envision the use of porch in embedded systems design that uses the C language for software development. Embedded systems are often built around commodity processors for which C (cross-) compilers and linkers exist or are relatively easy to port. One of the first steps of system software design is the implementation of I/O functions. The *ftIO* system assumes that a subset of the ANSI C file operations will be implemented.

Once a C compiler and I/O library are available, porch can be used for single-processor systems to capture the state of a computation including the files it accesses in a portable checkpoint. This checkpoint could be inspected on a binary incompatible machine, for example, by using a debugger on a workstation. The capability of porch to automatically generate code for converting data representations can simplify the design of multiprocessor systems, because machine-independent file data can be exchanged across binary incompatible processors without explicit coding for portability. We believe that porch and *ftIO* could be a viable alternative for software development, including debugging, as well as an aid for simplifying the software for fault-tolerant heterogeneous systems.

The paper is organized as follows. We first give a brief review of the porch compiler technology in Section II. In Section III we present the *ftIO* algorithm, which is in principle independent of the porch technology. Section IV discusses design alternatives for the implementation of *ftIO*. Experimental results of our prototype are presented in Section V. Related work is discussed in Section VI.

II. The porch Compiler

The porch compiler is a source-to-source compiler that translates C programs into equivalent C programs additionally capable of saving and recovering from portable checkpoints. *Portable checkpoints* capture the state of a computation in a machine-independent format, called *Universal Checkpoint Format—UCF*. The code for saving and recovering as well as converting the state to and from *UCF* is generated automatically by porch.

The porch compiler technology solves three key technical problems to render programs portable.

Stack environment portability: The stack environment is deeply embedded in a system, formed by hardware support, operating system and programming language design. A key design decision to implement porch as a source-to-source compiler has been the necessity to avoid coping with the system-specific state such as program counter or stack layout. It is not clear whether this low-level system state could be converted across binary incompatible machines. Instead, porch generates machine-independent source code to save and recover from checkpoints. At the C language level, variables can be accessed by their name without worrying about low-level details such as register allocation or stack layout done by the native compiler.

Data representation conversion: Two issues of data representations are of concern: bit-level representations and data layout. Basic data types are stored in different formats at the bit level. The most prominant formats are *little endian* and *big endian*. Furthermore, (memory) system designs require different alignments of basic data types. These determine the layout of complex data types such as structures. Consequently, all basic data types and the layout of complex data types are translated into a machine-independent formats. The porch compiler generates code to facilitate the corresponding conversions automatically.

Pointer portability: Pointers are rendered portable by translating them into machine-independent offsets within the portable checkpoint. Since the target address of a pointer is not known in general at compile-time, porch is supported by its runtime system to perform the pointer translation during checkpointing and recovery.

To enable code generation, *potential checkpoint locations* are identified in a C program by inserting a call to the library function `checkpoint()`. For these potential checkpoint locations, porch generates code to save and recover the computation's state from portable checkpoints.

The porch compiler generates code for programs that compile and run on the chosen target architectures without modifying the source code. This does not prevent system-specific coding such as conditionally compiled or hardware-specific code fragments, but requires structuring a program appropriately. The means of hiding system-specific details from porch are structuring the program into multiple translation units and functions. The porch compiler employs interprocedural analysis to instrument

only those functions of a program that are on the call path to a potential checkpoint location. Those functions that are not, 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. Only the portable portion of the code would be precompiled with porch. The nonportable portion could be different for particular systems, because it would not affect the state of the computation during checkpointing.

III. The *ftIO* Algorithm

This section describes the algorithm underlying the *ftIO* system. The *ftIO* algorithm is a private-copy/copy-on-write design for checkpointing systems, where phases of *normal execution* and *checkpointing* alternate unless a failure occurs. The *ftIO* algorithm is based on a finite automaton built upon a set of *ftIO* states for each file accessed by the application. State transitions occur if certain file operations are executed. 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* state.

We restrict the following discussion to a subset of the *ftIO* finite automaton. In particular, we omit the discussion of the file operations `remove`, `rename`, and those involving temporary files, as well as the associated *ftIO* states. The complete finite automaton can be found in [6].

A. *ftIO* States

Each file is associated with two orthogonal *ftIO* states. A file may be *dirty* or *clean*, and it may be *open* or *closed*.

clean/dirty A file is *clean*, if it has not been modified since the last checkpoint; otherwise, it is *dirty*. By this definition, files that do not exist are clean. In a private-copy implementation, no replicas exist for clean files.

open/closed A file can be either *open* or *closed*. By default, all files are considered closed unless they are explicitly opened. In particular, files that do not exist are closed by this definition.

Besides the file-specific states, a global boolean *ftIO* state is maintained.

FIN is a bit (for *finished*) used to record the success of the *ftIO* commit operation.

B. *ftIO* File Operations

We distinguish ANSI C file operations from *ftIO* file operations. The latter define the input alphabet of the *ftIO* finite automaton as listed below.

create creates and opens a file. If the file exists already, it is erased before being created again. This *ftIO* file operation corresponds to the ANSI C file operation `fopen` with opening mode `"w"`, or `"a"` if the file does not already exist.

open opens an existing file. The corresponding ANSI C file operation is `fopen` with opening mode `"r"`, or `"a"` if the file exists.

write modifies the contents of an open file, including appending data to that file. There are several corresponding ANSI C file operations: `fprintf`, `vfprintf`, `fputc`, `fputs`, `putc`, `fwrite`.

close closes an open file. This operation corresponds to the ANSI C file operation `fclose`.

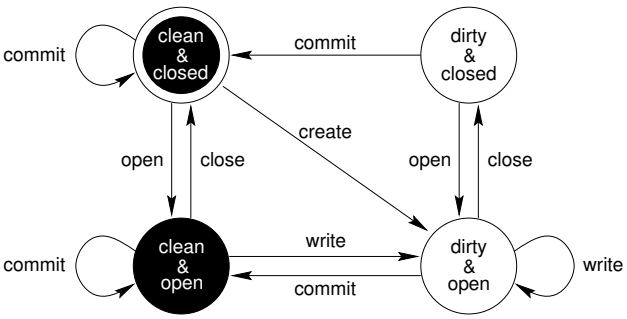

Fig. 1. Subset of transition diagram of the *ftIO* finite automaton. The start state is clean & closed; the accepting states after a commit operation are shown in black.

Figure 1 shows the transition diagram of the *ftIO* finite automaton based on the *ftIO* states and file operations described above. Opening and closing files does not affect the initial clean/dirty state. Open files can be written, causing the file to be marked dirty. In our *copy-on-write* implementation of *ftIO*, a clean file is copied upon a write, and the write as well as all future writes are performed on the replica. Upon file creation the dirty state is set.

Note that the open and close operations do not modify the contents of a file but affect a state transition. Furthermore, there is no *ftIO* file operation read. None of the ANSI C file operations `fscanf`, `fgetc`, etc. modify the contents of a file, and none of them imply a state transition in the *ftIO* finite automaton.

C. Checkpointing

We separate the checkpointing process into two phases. First, the porch runtime system (RTS) saves the volatile state including register and memory values. Second, the *ftIO* commit phase is responsible for handling files. The two phases are accompanied by a third action, the maintenance of the FIN bit. Figure 2 shows the sequence of the checkpointing phases.

Upon checkpointing, the volatile state of a process is saved by the porch RTS in *UCF* format in a temporary

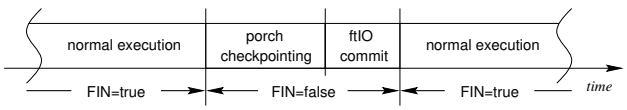

Fig. 2. Checkpointing phases.

checkpoint file. This checkpoint file includes the *ftIO*-specific flag FIN, explained in detail below. When the volatile state is gathered in the temporary checkpoint file, it replaces the previous checkpoint file by means of the atomic file operation `rename` [7]. After the porch RTS has saved the checkpoint, the *ftIO* system performs commit operations for all files. The commit operation affects only dirty files. For each dirty file, the original file is replaced by its replica, and the file state transitions to clean.

The FIN bit ensures the atomicity of the *ftIO* commit operation. It occupies a bit in the checkpoint. The porch RTS initializes a new temporary checkpoint file with the FIN bit value *false*. At the end of the *ftIO* commit phase, the FIN bit is set to *true* in the checkpoint file. This happens in the actual checkpoint file, which is the temporary file that replaced the previous checkpoint at the end of the porch checkpointing phase. Writing the FIN bit does not require an atomic disk-write operation. Instead, we only must ensure that the FIN bit has been written to disk before the first write operation after checkpointing.

D. Failure Cases and Recovery

Figure 2 above shows the two checkpointing phases embedded in periods of normal execution. Failures may strike during any of these phases. There are, however, only two distinguished failure cases, which simplifies reasoning about the correctness of the *ftIO* algorithm. Recovery is, analogous to the checkpointing process, split into two phases: During porch recovery, the volatile state of a computation is restored, and files are recovered during the *ftIO* recovery phase. Figure 3 illustrates the recovery phases.

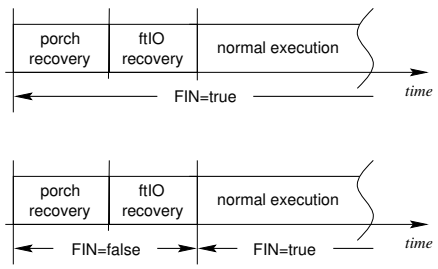

Fig. 3. Recovery phases if failure occured during normal execution or during porch checkpointing (top), and if failure occured during *ftIO* commit (bottom).

First, if the application aborts due to a failure during *normal execution* or during the porch *checkpointing* phase, recovery is based on the previous checkpoint. Since all modifying file operations have been performed on private copies, the original files are untouched. Because none of the modifications have been committed yet, recovery from the previous checkpoint and the original files is safe.

Second, if the application aborts during the *ftIO-commit* phase, only a subset of the files may have been committed before the failure. In this case, the previous checkpoint has been replaced with the new checkpoint already, and recovery will restore the computation from the new checkpoint. Files are restored simply by executing the *ftIO* commit operations during the *ftIO* recovery phase before returning control to the application. Since committing files involves no more than replacing the original files with their replicas, those files that have not been committed due to a failure can be replaced before the application restarts. Files that have been committed already remain untouched. The *ftIO* commit code does not even have to be changed, because the atomic `rename` operation is idempotent.

The FIN bit serves to distinguish the two modes of recovery. Figures 3 shows the values of the FIN bit for both modes; cf. Figure 2. In the following, we argue informally why the *ftIO* algorithm works due to the FIN bit without any atomic operations other than the `rename` file operation. Recall from the description above that the FIN bit is initialized to *false* within a new temporary checkpoint file. It is only changed to *true*, when the *ftIO* commit phase is finished. Note that writing the FIN bit into the checkpoint file is not atomic. The FIN bit must be on disk only before the first write operation after checkpointing, for example, using the UNIX operation `fsync`. Otherwise, recovery might commit write operations erroneously.

The FIN bit determines the restore action during the *ftIO* recovery phase. If the value of FIN in the checkpoint is *true* upon recovery, the checkpointing process succeeded, and the *ftIO* recovery phase is basically a `nop`. This case is shown at the top of Figure 3. Only if a failure occurs during the *ftIO* commit phase, does the FIN bit in the checkpoint remain *false*. In this case, files are committed during the *ftIO* recovery phase. No special treatment is required in case of failure after all files are committed but the FIN bit has not been set to *true* yet.

The only failure case remaining to be discussed is a failure during recovery, cf. Figure 3. No distinction is necessary as to whether the failure occurs during the porch recovery phase or the *ftIO* recovery phase. Since the *ftIO* recovery phase executes the idempotent rename code of the *ftIO* commit phase if FIN is *false* and is a `nop` otherwise, a failure during recovery is equivalent to a failure during checkpointing.

IV. System Design Considerations

In this section, we discuss design alternatives for *ftIO*. As a result of this discussion, a *copy-on-write* scheme appears to be the most reasonable concept for an implementation of *ftIO*.

A porchified program is a *transaction system* [8], where a transaction consists of the computation executed between subsequent checkpoints. Transactions are atomic operations. Either all operations within a transaction are executed, or none is. Transaction systems are generally classified as *undo-log* systems or *private-copy* (also called *side-file* [8]) systems. Undo-log systems record a log of undo operations for each operation within a transaction. If a failure occurs, the log is used to undo the modifications of an unfinished transaction. Private-copy systems maintain a private copy of all data accessed by the operations of a transaction and update the original only when the transaction commits. We briefly discuss the undo-log concept, the private-copy concept, and two possible implementations of the latter one, namely the *copy-on-write* and the *shadow-block* implementation.

A. The Undo-Log Concept

The undo-log concept assumes that a system fails rarely, and most transactions commit normally. Therefore, undo-log implementations modify the state of a system directly and keep a separate log to undo the modifications in case of a failure. If a transaction commits normally, the undo-log is simply discarded.

The undo-log concept is well suited for small transactions. If transactions are large, however, a serious time and space penalty is imposed. For each transactional operation, an undo record must be committed before the operation can be executed. Thus, runtime as well as storage requirements for the undo-log are proportional to the number of operations within a transaction.

In the context of file-I/O and checkpointed programs, the cost of the undo-log is proportional to the number of file operations that modify a file between subsequent checkpoints. The write operation is the most frequently occuring modifying file operation. For n writes between checkpoints, the runtime overhead is $\Theta(n)$ for committing the associated undo records, which require a space of order $\Theta(n)$. If no failures occur, committing a transaction involves no more than discarding the undo-log.

B. The Private-Copy Concept

In the private-copy concept, changes are not applied to the state of the system until a transaction is committed. Instead, a private copy is kept of the data, and all transactional operations are performed on the copy. During the commit phase of the transaction, the copy is reconciled with the original.

The space overhead of a private copy is bound by the size of the original file N plus potential bookkeeping overhead. The runtime overhead of the private-copy concept is bound by $O(N)$. The constant factor of this bound varies depending on the individual implementation. Note that both space and runtime overhead are bound by the file size N in the private-copy concept, whereas both are bound by the number of modifying file operations between checkpoints n in the undo-log concept.

In the following sections, we discuss implementations and system trade-offs that affect the performance of the private-copy concept in practice.

C. Copy-on-write Implementation

Copy-on-write is an implementation technique developed for virtual memory in the Mach operating system [5]. This technique can be applied in the conext of file-I/O, where it may be viewed as an implementation technique of the private-copy concept. In fact, our *ftIO* prototype is based on a copy-on-write implementation.

By copy-on-write implementation, we mean that a private copy of a file is made upon the first write operation after the last checkpoint or after the program begins execution or recovers, respectively. All subsequent file operations are performed on the replica until the next checkpoint is saved, when the original file is replaced with the replica. Copy-on-write is an optimization in the sense that a copy is only made, when the first write occurs, as opposed to copying at the very first access, which may be an open or read.

The copy-on-write implementation suffers runtime overhead only during the first write operation after a checkpoint. All other file operations incur virtually no overhead, with the exception of *ftIO*-related bookkeeping. During the commit or recovery phase, the original file is replaced by the replica. An atomic `rename` operation can be used to this end, as provided by POSIX-compliant operating systems [7].

Copy-on-write incurs no overhead, if a file is accessed by read operations only. It performs best in the presence of write operations whose modifying accesses are scattered across the entire address range of a file. In this case, write accesses are expected to amortize the copy overhead, because block-copying of files utilizes disk technology and memory bandwidth very efficiently.

The worst-case behavior of the copy-on-write implementation occurs if data are appended to a file. In this case, replicating the file could be avoided entirely. In fact,

the *ftIO* system implements an optimization for this case. Instead of copying a file, opened in append mode, the appended data are written to a new file, which is appended to the original file during the *ftIO* commit phase.

D. Shadow-Block Implementation

Replicating files may be considered prohibitive if either runtime is critical, such as in real time applications, or if the capacity of non-volatile storage is limited, as is often the case in embedded systems design. The shadow-block implementation [9], or shadow-page algorithm in [8], is an optimization of the copy-on-write implementation, which is suited for use in operating systems. Here, files are typically fragmented into disk blocks, and the file system maintains bookkeeping information about the fragmentation. In such an environment, rather than copying an entire file upon the first write, only the block accessed by the write is copied. Each block of a file can be copied, committed, and recovered independently.

However, the shadow-block implementation is not well suited for use in *ftIO*, which is implemented at user level for portability reasons. At user level, the fragmentation of files is invisible. Besides adding another level of fragmentation at user level, a shadow-block implementation in *ftIO* would cause additional overhead during the commit phase, as discussed in the following section.

E. Comparison of Implementations

This section compares the space and runtime overheads of the schemes discussed above. We are primarily interested in worst-case analysis, because it provides upper bounds on the overheads.

Let n be the number of write operations performed on a file, and let N be the number of bytes in the file. For the shadow-block scheme, let b be the number of blocks of the file and f the ratio of the modified blocks versus the number of blocks in the file ($0 \leq f \leq 1$). Writing to disk is assumed to require a startup time t_s and time d for each byte written. Thus writing n bytes takes time $t_s + nd$. Let D be the time to write an entire file: $D = Nd$. We compare the space overhead S and time overhead T of the different schemes assuming that n write operations are performed, and a fraction f of file blocks is modified.

For n write operations, an *undo-log* scheme yields $S = \Theta(n)$ and $T = n \cdot t_s$. The *copy-on-write* implementation incurs $S = N$ and $T = t_s + D$ for creating the replica, independent of n. The committing rename operation is not counted. The behavior of the *shadow-block* implementation is more complex. The space overhead is $S = fN$, with the worst case $S = N$ for $f = 1$, if all blocks are modified. The time overhead is $T = 2f(b \cdot t_s + D)$; two copies of $f \cdot b$ modified blocks are made. In the worst case, $f = 1$ and $T = 2(b \cdot t_s + D)$.

In the worst case, the undo-log is worse than the copy-on-write implementation. The space overhead of the undo-log scheme, $\Theta(n)$, is not bound by the file size N but by the number of file operations between checkpoints. In particular, $n \gg N$ could be a possible scenario. In contrast, the space overhead of the copy-on-write scheme is $S = N$. Also, the runtime overhead for committing n undo-logs, $T = n(t_s + \Theta(d))$, could be substantially larger than $T = t_s + D$ for the copy-on-write implementation. Furthermore, the worst case yields a higher runtime overhead for the shadow-block implementation than for the copy-on-write implementation: $2(b \cdot t_s + D) > t_s + D$, while both implementations imply $S = N$. As a result of this discussion, we implemented the copy-on-write scheme in the *ftIO* prototype, which is also substantially simpler to implement than the other two schemes.

Besides the worst-case analysis, it is instructional to compare the overheads of the copy-on-write and shadow-block implementations, as there exists a ratio of modified blocks $f_=$, where both schemes incur the same runtime overhead. For $f > f_=$, copy-on-write is faster, whereas for $f < f_=$, shadow-blocks are faster. We find that $T = 2f_=(bt_s + D) = t_s + D$ for $f_= = \frac{1}{2}\frac{t_s+D}{bt_s+D}$. Depending on the disk technology used and the file size, there are two boundary cases. If $b \cdot t_s \ll D$, both implementations incur the same runtime overhead for $f_= = 1/2$. For $b \cdot t_s \gg D$, we find that $f_= = 1/(2b)$. As a consequence, the shadow-block implementation is only preferable for relatviely small b, i.e. larger block sizes.

V. Experimental Results

We designed three synthetic benchmarks to evaluate the performance of our copy-on-write implementation of *ftIO*: a read benchmark, a write benchmark, and a random write access with spatial locality. These benchmarks are designed to exhibit the performance characteristics of *ftIO*. All runtimes were gathered on a Sun Sparcstation10 with checkpointing to local disk.

Figure 4 presents runtimes of the read benchmark and the write benchmark in the presence of checkpointing compared to the original, not porchified code. The read benchmark *reads* consecutive bytes of a file. The write benchmark *writes* (appends) bytes to a file. The runtimes reported are accumulated over the bytes read or written. The overhead of the checkpointed version of the *read*-benchmark is primarily due to checkpointing the program's state rather than due to the file operations. The checkpointed version of the *write*-benchmark, however, exhibits the expected overhead due to file replication upon

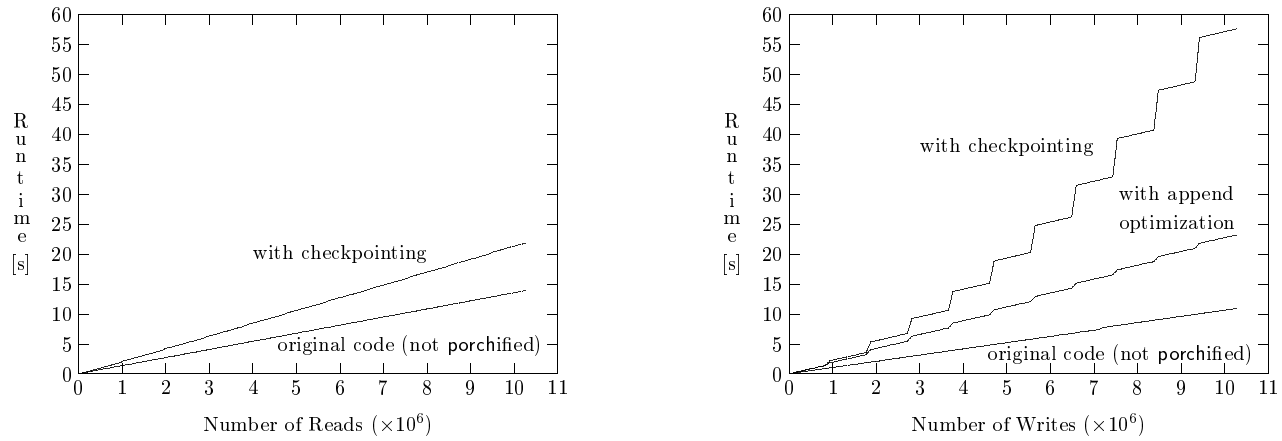

Fig. 4. Performance of reading (left) and appending (right) 10 Mbytes of data from and to a file. Checkpoints are taken after about 10^6 file operations in the experiments marked "with checkpointing" and those marked "with append optimization."

the first write after a checkpoint has been saved. Each step in that curve corresponds to the overhead of file replication. The monotonically increasing step size of the checkpointed version without append optimization is a result of the growing file size. Clearly, file replication dominates the checkpointing overhead. The curve of the checkpointed version with the *append optimization* exhibits a constant step size, which corresponds to checkpointing and copying the appended data from a temporary file to the original file. In the experiments, the same amount of bytes is appended between subsequent checkpoints.

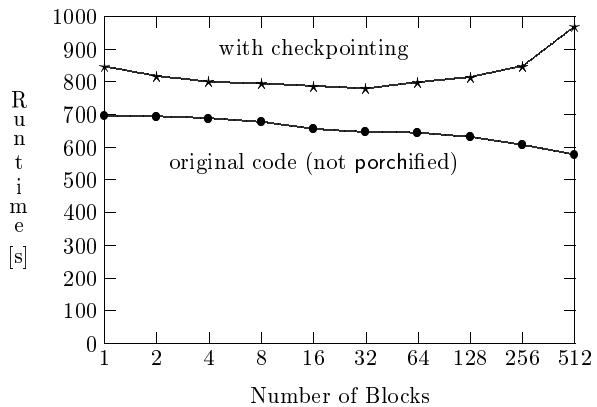

Fig. 5. Runtimes of random file accesses with spatial locality. The variation of the number of blocks simulates the behavior of a shadow-block implementation with different block sizes.

Figure 5 presents runtimes of our third benchmark. This benchmark analyzes the performance of *ftIO* with file operations randomly distributed over the address range of a file, and where file accesses exhibit spatial locality. Spatial locality is incorporated by computing the location of a file access to be normally distributed around the location of the previous file access with a standard deviation of 10.24 KBytes. This value corresponds to 0.1 % of the total file size of 10 MBytes. Our benchmark performs 10^6 file operations, 90 % of which are read operations, and 10 % are write operations. Ten checkpoints are taken at equal intervals during the execution of the benchmark.

Figure 5 includes runtimes of the copy-on-write implementation for different numbers of blocks. We simulate a shadow-block implementation by splitting the 10 MByte file into files (blocks) of equal size and accessing these blocks with our random strategy as if they occupied a contiguous address space. This simulation models a lower bound for the shadow-block implementation, because the copy-on-write implementation does not include explicit copying of blocks during the commit operation. Instead, each of the files (blocks) is renamed atomically.

The primary observation involves a comparison of the runtimes in Figure 5 with those in Figure 4. Accessing a file randomly (10^6 accesses) is almost three orders of magnitude more expensive than contiguous read or write accesses, even with spatial locality of the accesses. Secondly, the performance of the benchmark without checkpointing in Figure 5 improves as the number of files (blocks) increases. We attribute this behavior to the caching mechanisms in the Solaris operating system, which seems to be optimized for smaller files. Thirdly, the runtimes in Figure 5 indicate that the copy-on-write implementation is more reasonable than the shadow-block implementation. As the number of blocks increases from 1 to 32, performance improves by 8 % only. For larger numbers of blocks, performance degrades, primarily due to the cost of the rename operations.

VI. Related Work

We are not aware of any checkpointing system that implements portable, fault-tolerant file-I/O. According to Gray [8, p. 723], "a large variety of methods exists for

moving data back and forth between main memory and external storage." To our knowledge, the *ftIO* algorithm is new in that it uses only an atomic rename operation and a single state bit to implement transactional file operations.

The libft checkpointing library [10] has been extended to provide transactional file operations in the context of checkpointed applications. This work implements the undo-log concept, using a two-phase commit protocol to commit a transaction. Besides requiring explicit denotation of a program to mark the start and end of a transaction, libft does not provide portability.

A number of checkpointing systems exist for homogeneous environments, such as Libckpt [11] or a supplement of the Condor system [12]. Not only are these systems tied to binary compatible machines, they also do not support transactional file operations.

There are two approaches to migration across binary incompatible machines, the Tui system [13] and the work on Dynamic Reconfiguration [14]. Neither of these systems supports transactional file operations either.

For implementing *ftIO*, we also considered the *twin-diff* technique, which is popular for building distributed shared memory systems [15]. Our analysis revealed that the worst-case runtime overhead of a twin-diff implementation is even worse than that of the shadow-block implementation due to the additional diffing computation.

The work on I/O benchmarking by Chen and Patterson [16] inspired the development of our synthetic benchmarks for transactional file-I/O.

VII. Conclusion

We present the *ftIO* system, which extends the porch compiler by providing portable, fault-tolerant file-I/O. Due to the *ftIO* system, C programs that use formatted file-I/O can be rendered fault-tolerant in a transparent fashion by precompilation with porch. Furthermore, no changes are necessary to the system-specific C library. We introduce the *ftIO* algorithm, which provides transactional file operations using an atomic rename and a single state bit only. The *ftIO* code itself is compiled with porch, thereby ensuring that a checkpoint comprises *ftIO*-state information in a machine-independent format. We discuss and analyze the design choices for *ftIO* and present experimental data of our prototype implementation.

VIII. Acknowledgement

This work has been funded in part by DARPA grant N00014-94-1-0985. We thank Charles Leiserson for his support and proof reading. Matteo Frigo contributed comments on an early draft of the paper.

References

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

[2] Volker Strumpen, "Compiler Technology for Portable Checkpoints," submitted for publication (`http://theory.lcs.mit.edu/~strumpen/porch.ps.gz`), 1998.

[3] Stuart I. Feldman and Channing B. Brown, "Igor: A System for Program Debugging via Reversible Execution," *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, vol. 24, no. 1, pp. 112–123, Jan. 1989.

[4] Herbert Schildt, *The Annotated ANSI C Standard*, McGraw-Hill, 1990.

[5] Richard Rashid, Avadis Tevanian Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 896–908, Aug. 1988.

[6] Igor Lyubashevskiy, "FTIO: A System for Fault-Tolerant File-I/O," M.S. thesis, MIT Department of Electrical Engineering and Computer Science, May 1998, In preparation.

[7] Peter J. Plauger, *The Standard C Library*, Prentice Hall, Englewood Cliffs, 1992.

[8] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, 1993.

[9] Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, 1992.

[10] Y. M. Wang, P. Y. Chung, Y. Huang, and E. N. Elnozahy, "Integrating Checkpointing with Transaction Processing," in *Digest of Papers—27th International Symposium on Fault-Tolerant Computing*, Seattle, Washington, June 1997, pp. 304–308, IEEE Computer Society.

[11] James S. Plank, Micah Beck, and Gerry Kingsley, "Libckpt: Transparent Checkpointing under Unix," in *USENIX Winter 1995 Technical Conference*, New Orleans, Louisiana, Jan. 1995, pp. 213–233.

[12] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System," Tech. Rep. 1346, Computer Sciences Department, University of Wisconsin-Madison, Apr. 1997.

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

[14] Christine Hofmeister, *Dynamic Reconfiguration*, Ph.D. thesis, Computer Science Department, University of Maryland, College Park, 1993.

[15] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad, "Software Write Detection for a Distributed Shared Memory," in *1st Symposium on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994, USENIX, pp. 87–100.

[16] Peter M. Chen and David A. Patterson, "Storage Performance—Metrics and Benchmarks," *Proceedings of the IEEE*, vol. 81, no. 8, pp. 1151–1165, Aug. 1993.