Compiler Support for Fine-Grain Software-Only Checkpointing

Chuck (Chengyan) Zhao[1], J. Gregory Steffan[1],
Cristiana Amza[1], and Allan Kielstra[2]

[1] Department of Electrical and Computer Engineering, University of Toronto
{czhao,steffan,amza}@eecg.toronto.edu
[2] IBM Canada Toronto Laboratory
kielstra@ca.ibm.com

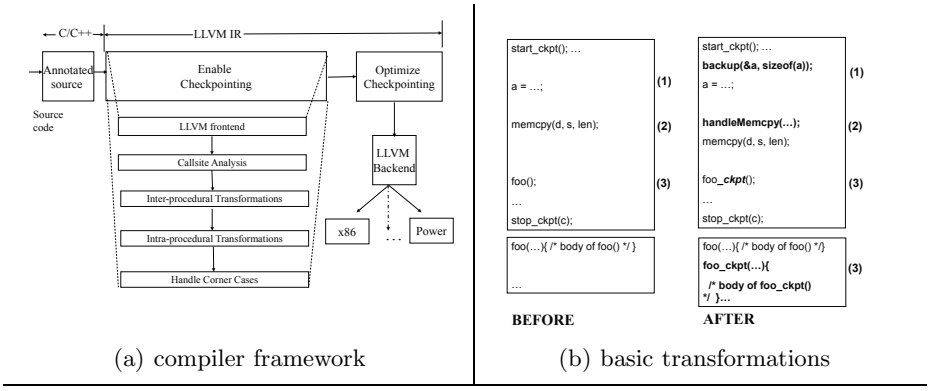
Abstract. Checkpointing support allows program execution to roll-back to an earlier program point, discarding any modifications made since that point. Existing software-based checkpointing methods are mainly libraries that snapshot all of working-memory, and hence have prohibitive overhead for many potential applications. In this paper we present a light-weight, fine-grain checkpointing framework implemented entirely in software through compiler transformations and optimizations. A programmer can specify arbitrary checkpoint regions via a simple API, and the compiler automatically transforms the code to implement the checkpoint at the granularity of individual stores, optimizing to remove redundancy. We explore two application areas for this support. First, we investigate its application to debugging, in particular by providing the ability to rewind to an arbitrarily-placed point in a buggy program's execution. A study using BugBench applications shows that our compiler-based approach is more than 100x less overhead than full-process checkpointing. Second, we demonstrate that compiler-based checkpointing support can be leveraged to free the programmer from manually implementing and maintaining software rollback mechanisms when coding a back-tracking algorithm, with runtime overhead of only 15% compared to the manual implementation.

1 Introduction

Checkpointing [7,16,19,24,27,30,31] is a technique to back-up program state such that execution can later revert to the backup, to recover from program failure or mis-speculation. While proposed hardware-based checkpointing solutions [4,13] show promising performance, they are not yet available in commodity systems. Software-based checkpointing solutions [16,19,25,31] can be used on commodity hardware, but can also have prohibitive overheads as they are typically coarse-grained, meaning that they back-up large ranges of memory if not the entire process image.

In this paper we propose a software-only method for checkpointing program execution that is implemented in a compiler. In particular, our transformations implement checkpointing at the level of individual variables, as opposed to previous work that checkpoints entire ranges of memory or entire objects [4,7,16,25].

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 200–219, 2012.



(a) compiler framework (b) basic transformations

Fig. 1. Framework and basic transformations

The intuition is that such fine-grain checkpointing can (i) provide many opportunities for optimizations that reduce redundancy and increase efficiency, and (ii) facilitate uses of checkpointing that demand minimal overhead. We present a complete checkpointing framework and optimization infrastructure that can (i) enable software-only checkpointing over arbitrarily large and complex program regions and (ii) leverage compiler optimizations to reduce overhead. We show that our fine-grain scheme is more efficient than coarse-grain approaches, and that up to 98% of checkpoint buffer space and up to 95% of `backup` memory calls can be eliminated.

We demonstrate the utility of our compiler-based checkpointing infrastructure via two different applications of this support. The first is support for debugging, in particular by giving the programmer the ability to roll-back execution to repeatedly examine the state of a program prior to the manifestation of a bug. We study several flawed applications from the BugBench [20] suite and demonstrate the low overheads of checkpointing support for rollback. The second is support for back-tracking algorithms, where the programmer can avoid manually implementing support for rewinding data-structures, instead leveraging compiler-based checkpointing to provide it automatically. We study VPR [5,6], in particular the simulated-annealing-based place-and-route algorithm for FPGAs, which optimistically swaps blocks and either keeps or discards the swap depending on whether a cost function is improved. We compare the original manual implementation of back-tracking support to our automatic compiler-based approach.

2 Basic Checkpointing

Figure 1(a) presents an overview of our checkpointing system, implemented as passes in the `LLVM` [17,18] compiler infrastructure. It takes as input a source program with programmer annotations, and outputs transformed `LLVM IR` code that can target the multiple native platforms that LLVM supports. LLVM provides a `C` back-end that allows the conversion of optimized IR back to `C` source code.

This source-to-source approach allows us to capitalize on all of the optimizations of the back-end compilers.

Programmer Interface. We assume a very simple programmer interface to checkpointing: the user delimits the desired checkpointing region via the interface calls `start_ckpt()` and `stop_ckpt(c)`, where `c` is a boolean variable that indicates whether the checkpoint should be rewound/re-executed or committed. The compiler then instruments all relevant write operations with `backup` calls, each taking as arguments a pointer to the destination's address and its size in bytes. These `backup` calls are later optimized and inlined, but for now we show them in the code for illustration.

Callsite Analysis. Our compiler needs to know all user-defined functions that may be called directly or indirectly from the checkpoint region. We call the process of discovering such functions *callsite analysis*. The callsite analysis visits each node in the application's sub call graph originated from the annotated checkpoint region. It recursively identifies all user-defined functions in this partial call graph and marks them as requiring the creation of a checkpoint-enabled version.

Intra-procedural Transformation. The compiler then converts code in the user-annotated region into its checkpoint-enabled version in three steps. Step 1 is to precede each write with code to backup the write. Figure 1(b)(1) shows that variable `a` is modified and thus preceded with a `backup` operation. Step 2 is to handle certain system functions that have implicit memory writes. Figure 1(b)(2) illustrates the handling of one such routine, `memcpy`, by placing a handling function, `handleMemcpy`, immediately before it. Step 3 is to rename any user-defined function callsite within the region. Figure 1(b)(3) shows that a user callsite `foo` is renamed to its checkpoint-enabled version, `foo_ckpt`.

Inter-procedural Transformation. The final step is to enable checkpointing on all user-defined routines that are identified through the callsite-analysis phase. For each identified function, we clone its function body and rename it by appending `_ckpt` to its name, as shown in Figure 1(b)(3). Within the body of the cloned function, we recursively and repetitively apply the same three actions introduced in `Intra-procedual Transformation` above. In the end we produce a checkpoint-enabled version for every user-defined function that can potentially be called from the checkpoint region.

Handling Function Pointers. Since our checkpointing scheme clones user-defined functions, the compiler needs to identify the precise callee function at compile time. However, calls via function pointers might only be resolved at runtime. As shown in Figure 2(a), we handle this function pointer ambiguity by changing from a function pointer call to a normal function wrapper call with function pointer arguments. Within the wrapper function, each possible callee is explicitly examined through a list of parameter-matched candidates.

Handling Early Exits. Another special case deals with early exits from the checkpointing region, as shown in Figure 2(b). A `return` within the checkpoint

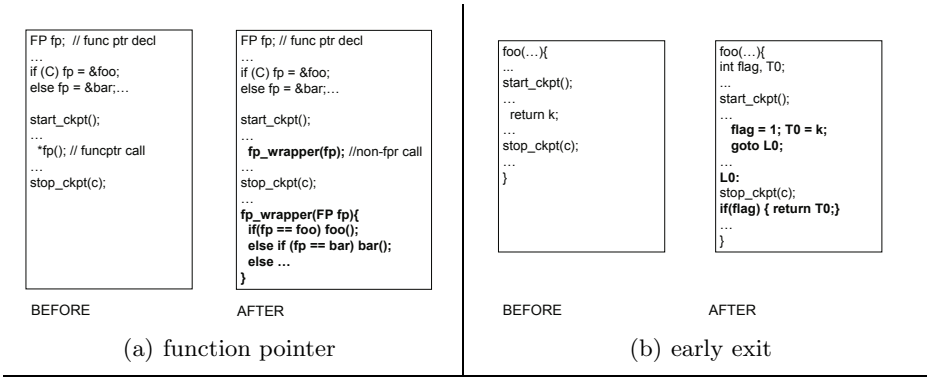


Fig. 2. Examples of handling function pointers and early exits

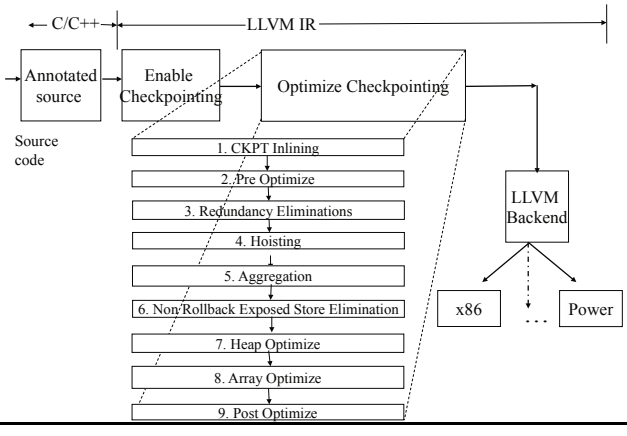


Fig. 3. Overview of optimization passes

region may prematurely terminate the checkpoint process without visiting the `stop_ckpt` marker. This violates the rule that the checkpoint region markers must be visited in pairs. Figure 2(b) suggests a possible solution: code is transformed to have a `goto` that branches to the `stop_ckpt` marker and reserves the appropriate return value.

3 Optimizations

Base transformations enable checkpointing on any user-annotated region by backing up the memory contents before each explicit or implicit write. This creates a large number of `backup` calls that are potentially redundant and leaves ample opportunities for optimization. Figure 3 provides an overview of our checkpointing optimization framework, that takes as input the checkpoint-enabled code produced as described in the previous section. The framework includes

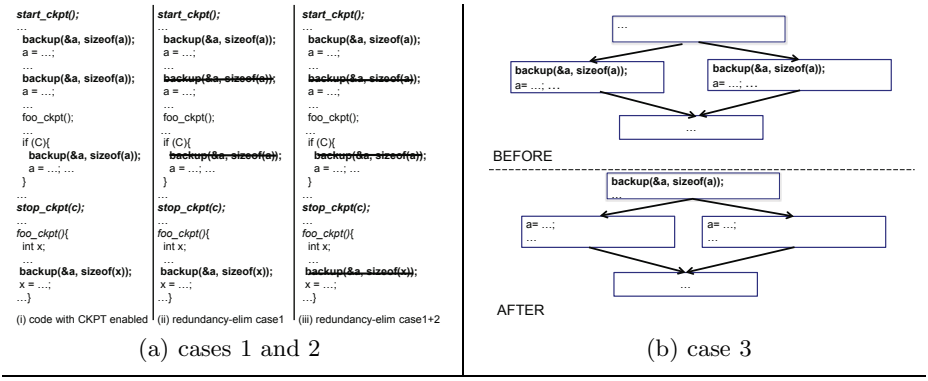


Fig. 4. Redundancy elimination cases 1-3 via code examples

more than 10 different optimizations and we introduce them in order of importance.

Redundancy Elimination. The most important optimizations are three cases of redundancy eliminations (*RE*1, *RE*2, and *RE*3), as illustrated in Figure 4. *RE*1 uses *dominating* relationships among `backup` calls. It identifies all `backup` calls with the same address and length that dominate the `stop_ckpt` region marker (e.g., the first three `backup` calls in Figure 4(a)), establishes the first in the sequence as the *leading* `backup` call, and then removes any remaining ones that are dominated by the leader. *RE*2 identifies all `backup` operations on a function's non-pointer-type local variables (i.e., the fourth `backup` call in Figure 4(a)). Since local variables are allocated on the stack and have no memory footprint in its enclosing function's calling context, it is safe to remove backups on local variables within any checkpoint-enabled function without impacting the correctness of checkpointing. *RE*3 performs similarly to common sub-expression elimination (*CSE*) by finding duplicate `backup` operations on both sides of a branch (as shown in Figure 4(b)). Once it finds a suitable pair, it hoists one of the backup calls into the immediate dominator block, and removes the other backup call.

Hoisting. Hoisting optimization aims to reduce redundant backup calls within loops (as illustrated in Figure 5(a)), by hoisting the backup of any variable written unconditionally within a loop to the loop header (e.g., variable `z` in the example). Such hoisting would not be performed by a normal compiler hoisting pass since the write to the variable is not necessarily loop invariant. The decision to hoist conditionally-modified backup calls is a trade-off, the conditional code must be executed frequently enough to be worth the cost of the non-conditional backup call in the hoisted version. Through experiment we found that it is generally not worth hoisting such conditionally-modified variables, at least not without profile feedback as a guide. To illustrate, in the example we choose not to hoist variable `y`.

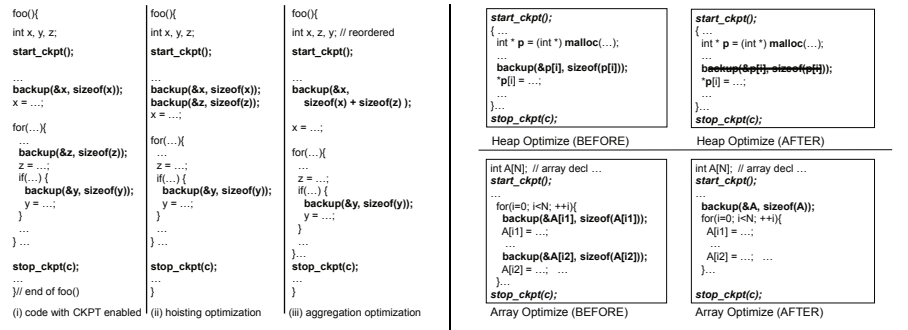


Fig. 5. Examples of hoisting, aggregation, heap, and array optimizations

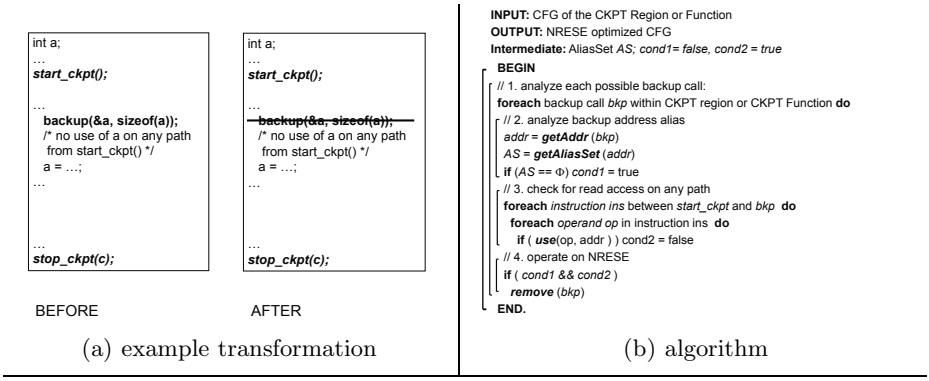
Aggregation. Aggregation examines `backup` calls for variables that are adjacent in memory, potentially rearranging the layout of the variables to ensure that they are adjacent. Figure 5(a)(iii) shows that two individual `backup` operations on variable `x` and `z` can be merged into a single one, covering the entire memory range for both variables.[1] Aggregation reduces the overhead of managing adjacent variables individually.

Dynamic Memory Optimization. Opportunities exist for any `backup` call that operates on dynamically allocated (heap) memory. If the heap allocation site is within the checkpoint region and it dominates the write, the `backup` operation on this write into heap-allocated memory can be eliminated. Figure 5(b) demonstrates the process of removing a `backup` on heap-allocated variable `p[i]`. Since the heap allocation happens within the checkpoint region, the heap-allocated contents have no memory footprint before checkpoint starts. Hence such backup calls can be eliminated since they are unneccesary.

Array Optimization. More interesting cases occur among backup operations on writes to array-based data inside a loop, as shown in Figure 5(b). Both writes into `A[i1]` and `A[i2]` are correlated with loop index variable `i`. It could be beneficial to merge multiple backups on individual array elements into a single backup operation, potentially covering a continuous array sub-range or even the entire array. We develop an algorithm that considers not only the array size, loop trip count and store intensity, but also a tolerance factor that a user can control through command-line options. Non-continuous array writes may happen when the program executes inside the loop, thus the tolerance factor specifies the trade-off in buffer-space used versus performance.

Non-rollback-Exposed Store Elimination. Given any variable that is written inside the checkpoint region, if there is *no* read of that variable on any path

[1] Note that for a source-to-source transformation this isn't necessarily a safe optimization as the back-end compiler may further rearrange the variable layout—an implementation in a single unified compiler would not have this problem.



(a) example transformation (b) algorithm

Fig. 6. Non-rollback-exposed store elimination optimization (NRESE)

from the beginning of the region, and its address has no alias, then an optimization can remove the respective `backup` operation for this variable without impacting checkpointing correctness. We call this optimization *non-rollback-exposed store elimination* (*NRESE*). Figure 6(a) shows an example of NRESE. Notice that the `backup` operation on variable `a` can be safely removed, since there are no direct or aliased reads of `a` along any path from the beginning of the checkpoint region. The value of `a` is recomputed each time and this re-computation is essentially independent of the current value of `a`. The algorithm presented in Figure 6(b) relies on performing an alias analysis to that `a` has no alias—we use the basic alias analysis (basic_aa) provided with `LLVM`.

Miscellaneous Optimizations. Inlining is applied to all remaining `backup` operations, allowing later standard optmiizations to schedule and optimize the contained instructions. `Pre-Optimize` and `Post-Optimize` passes perform miscellaneous clean-up operations, such as removing zero-length `backup` calls).

4 Buffering Implementation

The most important design decision in a checkpointing scheme is the approach to buffering: whether it will be based on *write-buffering* [11,21] or *undo-logging* [14,23]. A write-buffer approach buffers all writes from main memory, and therefore requires that the write-buffer be searched on every read. Should the checkpoint commit, the write-buffer must be committed to main memory; should the checkpoint fail, the write-buffer can simply be discarded. Hence for a write-buffer approach the checkpointed code proceeds more slowly, but with the benefit that parallel threads of execution can be effectively checkpointed and isolated (e.g., for some forms of optimistic transactional memory [11,22]). An undo-log approach maintains a buffer of previous values of modified memory locations, and allows the checkpointed code to otherwise read or write main memory directly. Should the checkpoint commit, the undo-log is simply discarded; should the checkpoint fail, the undo-log must be used to rewind main memory. Therefore an undo-log approach

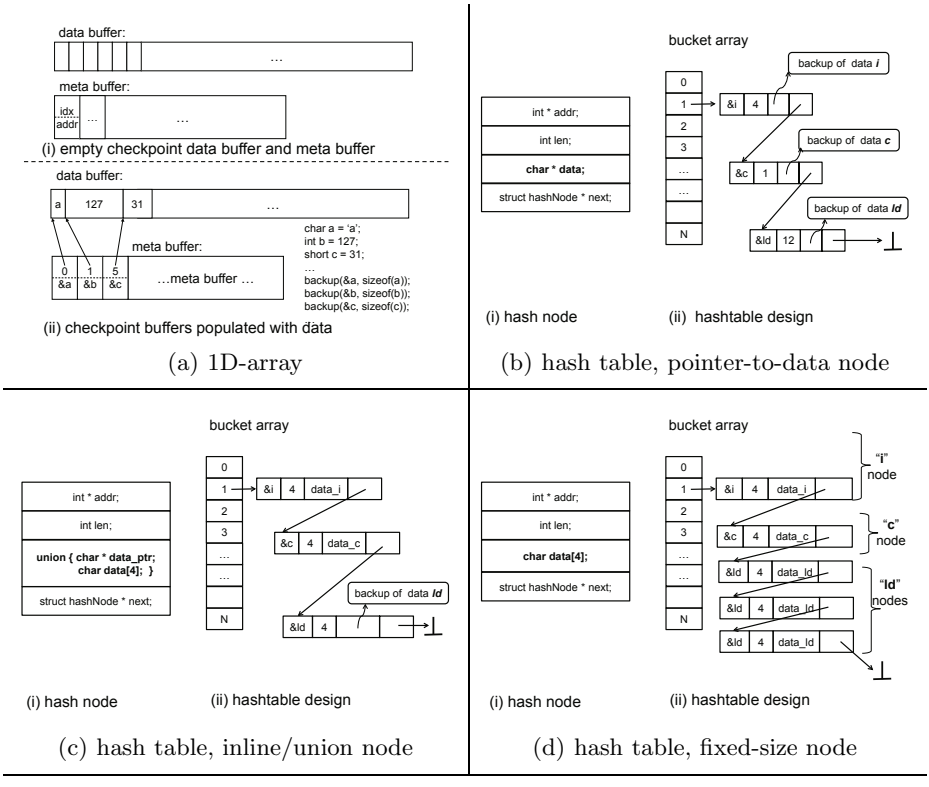


Fig. 7. Design options for an undo-log implementation

is best for the case of single-threaded code where checkpoint-rewind is uncommon, hence we focus solely on an undo-log approach for the remainder of this paper.

Figure 7(a)(i) illustrates a straightforward design of an undo-log based on the use of 1D arrays, where we have divided the undo-log buffer into two structures: (i) an array that is a concatenation of all backed-up data values of arbitrary sizes; and (ii) a meta-data array that stores the length and starting address of each element. As an example, Figure 7(a)(ii) shows the contents of an undo-log after three `backup` calls. When a checkpoint commits, we simply move the data and meta-data pointers back to the start of each array; when a checkpoint must be rewound, we use the meta buffer to walk backwards through the data buffer, writing each data element back to main memory.

While simple, a 1D-array-based undo-log suffers from redundancy, as each new backup call simply appends a value to the log without searching for an existing entry for that location; to search the array linearly would be prohibitively expensive. An alternative is to use a hash-table to allow fast search of prior entries for matches, to eliminate all redundancy in the undo-log. There will be a trade-off in the performance savings of reduced storage (due to reduced redundancy) versus the performance cost of hash-lookups.

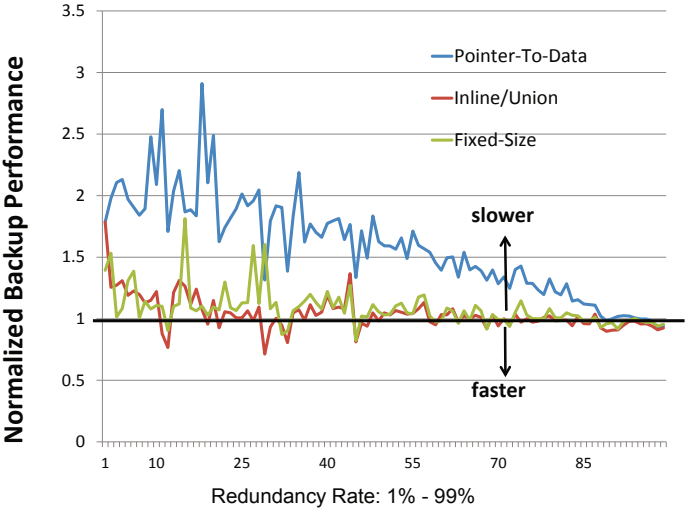


Fig. 8. Performance impact of four different buffer schemes over a wide range of redundancy rates. The x-axis represents redundancy rate from 1% to 99%; the y-axis is the relative checkpoint performance normalized to using a 1D-array. The figure represents checkpoint buffer with 1024 unique backup addresses, with only 4-byte backup length.

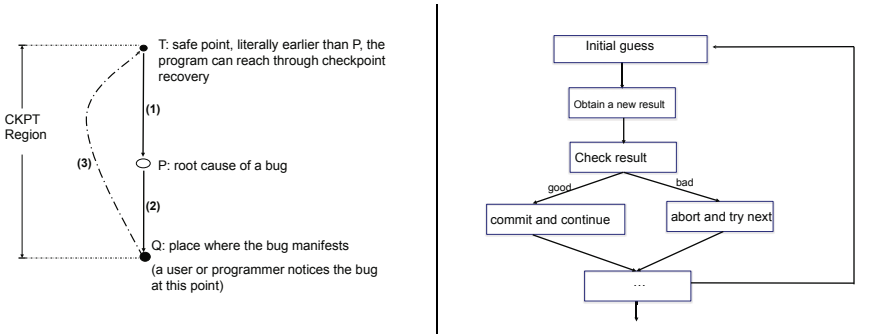
Hence we consider three hash-table designs, as illustrated in Figure 7, based on the options for the design of a hash table node: *pointer-to-data* (*PTD*), that stores a pointer to dynamically-allocated data storage; *inline/union* (*union*), that stores a union field that can be used either to directly store a 32-bit value inline, or instead as a pointer to dynamically-allocated data storage larger than 32 bits; and *fixed-size* (*fixed*), that always stores 32 bits of data per node and requires a list of nodes to store larger data values.

To compare the potential undo-log implementations we measure their *redundancy rate*, defined as follows. Let $Access(R)$ denote the total number of backups of a particular variable R that is written at least once within the checkpoint region, then the redundancy rate (RR) for this region can be defined as

$$RR = \frac{\sum_1^n (Access(R_i) - 1)}{\sum_1^n Access(R_i)} \tag{1}$$

where n is the total number of unique addresses that are checkpointed within the region. RR quantifies the amount of checkpointing redundancy as a floating point value between 0 and 1. In an ideal region where each unique variable address is checkpointed exactly once, its RR rate will be 0. The higher the RR rate, the more redundancy remains.

In Figure 8 we evaluate the trade-offs between the four buffering implementations above on microbenchmarks. We vary the microbenchmark access patterns to produce redundancy rates that vary from 1% to 99%, and report



(a) debugging (b) back-tracking, for VPR

Fig. 9. Overview of applications enabled by fine-grained checkpointing support

checkpoint performance normalized to that of using a 1D array. Overall, the solution based on a 1D array almost always outperforms all hashtable-based solutions. All three curves converge at a very high *RR* rate (close to 99%). With increasing redundancy rates, the performance difference among different backup schemes diminishes. The three different hashtable-based implementations have perfect storage behaviors; however this comes at a performance cost, mainly due to the poor cache locality of their link-list accesses. *Union* and *fixed* are both heavily optimized for dynamic memory management, thus their performance is considerably and consistently better than *PTD*. In summary, because of its superior performance, we focus on the 1D-array implementation of the undo-log for the remainder of this paper.

5 Checkpoint-Enabled Applications

Our compiler-based fine-grained checkpointing scheme can be leveraged in a wide range of applications. In this section, we introduce two important application domains that can benefit by either gaining additional functionality or through a simplified programming interface: checkpoint support for debugging, and checkpoint-enabled automatic back-tracking.

5.1 Checkpoint Support for Debugging

Program debugging is used to identify and resolve software bugs. A normal debugging session begins with user placing breakpoints at multiple pre-determined program locations, and stops execution at each location to examine the program's logic and states. However, once execution passes a certain breakpoint, it is normally difficult to rewind execution to a previous location though a user may often find that the root cause of a bug is likely located close to a previous breakpoint. Frequently restarting execution can be impractical because it may take a long time to reach the suspicious bug location.

Debuggers enhanced with our checkpointing support can help alleviate this situation. We expose the checkpoint APIs on the source-code level so that a programmer can selectively mark a checkpoint region that likely contains the bug, as shown in Figure 9(a). The programmer first inserts a end-region marker slightly after the bug-trigger location. Properly identifying a start-region position requires some understanding of the code. The region needs to be big enough to contain the root cause of the bug, but can't be too big so that the programmer is lost in unrelated details. In practice, we often place breakpoints overlapping with the checkpoint region boundaries. Once execution reaches the end of the region, the programmer decides whether he wants to finish debugging this region (by issuing a `commit_ckpt` command), or rewind and re-examine the current region (by issuing an `abort_ckpt` command).

Debuggers with our checkpointing support can rewind execution to a previously identified program location and re-examine the program region with unlimited number of retries. There is no restriction on the size of the region because we checkpoint into main memory and can dynamically grow the checkpoint buffer when needed. Eliminating program restart not only avoids all problems related with non-deterministic execution and availability of input, but also helps to reduce debugging cycle time. In practice we find it easy to use such rewind-capable debugger. The restart-free debugger with checkpointing support leads to shorter debugging cycle – allowing a programmer to rapidly identify root causes of a bug, thus converting the checkpointing capability into improved productivity.

5.2 Checkpoint Support for Automated Back-Tracking

Back-tracking refers to a set of algorithms that search for solutions in a given space of possible choices. A partial result may be either committed or discarded, depending on the evaluation result from it.

We study Versatile Placement and Route (*VPR*) [5,6], a CAD tool for generating high-quality circuit layouts on array-based FPGAs. VPR places and routes on a wide variety of FPGAs and facilitate comparisons among different architectures. VPR implements a software back-tracking algorithm in its placement phase, as shown in Figure 9(b). The algorithm starts as its input a randomly generated guess. It evaluates the result based on this attempted input. If the result is positive, it will be incorporated into the current system. Otherwise, the negative result is discarded. This process continues until a desired terminating condition is satisfied. Current implementation of VPR saves all necessary program states before attempting a new input. Shell a discard happen, it manually restores all saved program states from various complex data structures. VPR designers need to understand not only the placement algorithms, but also pay close attention to details of manually save and restore necessary program states. This is a tedious and error-prone process that often has a negative impact on productivity, especially when improving the algorithm that results data structure changes.

By exposing the checkpoint APIs at the source-code level, our fine-grain checkpoint framework frees VPR from details of conducting back-tracking operations.

Table 1. Benchmarks and Checkpoint Region Properties

Apps	Region	avg insts	avg source lines	entries
bc-1.05	S	2.2 K	3	3
	M	208 K	430*	1
	L	305 K	1200*	1
gzip-1.24	S	0.9 K	1	1
	M	2.7 K	89*	1
	L	194 M	119*	1
man-1.5h1	S	1.4 K	14	1
	M	1.6 K	30*	1
	L	645 K	89*	1
ncompress-4.2	S	0.8 K	2	1
	M	149 K	149*	1
	L	231 K	163*	1
polymorph-0.4.0	S	1.5 K	2	1
	M	3.1 K	49*	1
	L	148 K	76*	1
VPR-5.02		67.1 K	268*	371 K

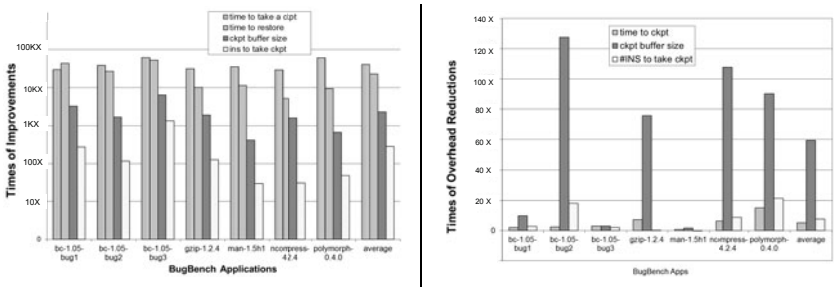
VPR designers can ignore all details of manual checkpointing and instead call `abort_ckpt()` or `commit_ckpt()`, which performs checkpoint abort and commit actions respectively. The simple APIs enable automatic software back-tracking on VPR, as well as all applications that have a need to perform back-tracking. VPR designers can instead focus on improving the algorithm – an step that simplifies application programming interface and improves end-user productivity.

6 Evaluation

In this section we evaluate our fine-grain software-only checkpointing framework. Our compiler infrastructure builds on the LLVM [17,18] open-source compiler infrastructure release 2.9—all analyses, transformations, and optimizations are organized as LLVM passes. For debugging support we consider Bugbench [20] applications, a suite containing various known software bugs plus program inputs that trigger them; we select five BugBench applications that contain buffer-overflow bugs. To evaluate back-tracking support we study a recent version of VPR-5.02 [5], as described in Section 5.2. We measure on an Intel Core i7 920 CPU, with 4GB of DDR3 RAM, running Debian6-i386 with `g++` version 4.4.5.

6.1 Checkpoint Region Selection

Table 1 summarizes the checkpoint regions for each benchmark application. For the selected applications from the BugBench suite, we enclose the root cause and manifestation of each bug in a minimal checkpoint region called the *small (S)* region. We then grow the small region by both forward-and-backward extending



(a) our approach relative to coarse-grain libCKPT

(b) our approach relative to runtime-based ICCSTM

Fig. 10. Overhead reduction relative to conventional checkpointing methods for BugBench applications

the region boundaries, covering increasing granularity and complexity of the source code. The result is a *medium (M)* region that contains a significant portion of the program, and a *large (L)* region that can potentially cover the entire application. VPR has only one checkpoint region as appropriate for properly implementing back-tracking within the `try_swap` function, although we have two implementations, *medium (M)* and *large (L)*, depending on whether the region is marked from the function callee's perspective or the caller's perspective, respectively. Checkpoint regions are vastly different in size: for example, a small region usually contains around 1000 instructions and spans 2–3 lines of source code, while a large region can contain up to 195 million instructions (e.g., gzip-1.24) and covers 1000+ lines of source code (e.g., bc-1.05)[2].

6.2 Comparison with Existing Checkpointing Solutions

In this section we compare our compiler-based checkpointing solution with two alternative software approaches to checkpointing: a checkpointing library, and a software transactional memory library supported by a commercial compiler.

Library-based schemes back-up all of the memory used by the running process—thus the checkpointing overhead closely correlates to the size of memory at checkpointing time. We use libCKPT [26] as the representative of a library-based software checkpointing solution. Figure 10(a) shows that our fine-grained checkpointing approach provides over 1000X overhead reduction compared to coarse-grain checkpointing, for both the time-to-take a checkpoint and the time-to-restore a checkpoint. The corresponding improvement in terms of the checkpointing metrics of checkpoint buffer size and the number of instructions needed to service a checkpoint are within the range of 100X to 1000X.

[2] Note that *M* and *L* regions always contain user-defined functions, thus the number of source lines presented in Table 1 marked with * only indicates the lower bound of possible source-code span.

We further compare software overheads for supporting single-threaded speculative optimization in Intel's Software Transactional Memory (STM) [1,28] (*ICCSTM*) versus our compiler-based checkpointing solution. (*ICCSTM*) is a software solution for supporting optimistic parallelism, based on Intel's production-quality `C/C++` compiler. Just like other STM systems, *ICCSTM* supports speculative parallel execution through write-buferring and dependence tracking of the reads and writes of multiple threads at run-time. The differences in performance between the two software packages are expected to come from the different focus and specialization on their respective main use cases.

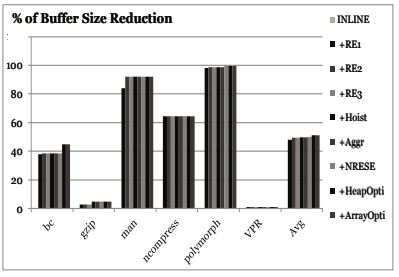
ICCSTM is mainly optimized to support program parallelization based on relatively short transactional regions. On the other hand, our checkpointing software is optimized to support single-thread speculation, or debugging for larger program regions. Based on the limited description available [1,28], *ICCSTM* uses only basic compiler optimizations such as inlining and a very simple form a partial redundancy elimination. Furthermore, to the best of our knowledge, *ICCSTM* does not optimize for the single-threaded speculative execution case. In this special case of speculation support, tracking of the single thread's read-set could be safely omitted. In contrast, our checkpointing scheme benefits from being specialized for the single-thread case. Specifically, we track only the write set for the speculative thread via an efficient implementation based on undo-logging. In the common case where speculation is successful, undo-logging avoids expensive lookups on reads for matching prior writes, and also the copies of writes to shared memory on commit. Overall, it is expected that our fine-grain checkpoint support will have lower overheads, and/or better cache behavior than a write-buffering STM.

Figure 10(b) compares *ICCSTM* to our baseline compiler-based checkpointing solution (with no optimizations). We find that our solution outperforms *ICCSTM* in almost all cases. On average, our solution outperforms the time-to-take a checkpoint for *ICCSTM* by 5X, and the number of instructions needed to take a checkpoint by 8X. The largest difference is in terms of the checkpoint buffer size, which is almost 60X lower for our solution.

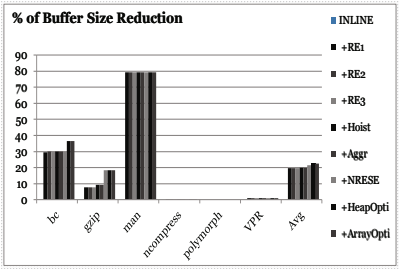
6.3 Optimization Effectiveness

To evaluate our checkpointing optimization framework, we run optimizations over each application's *M* and *L* regions. We gradually increase the number of optimizations on each test region until all available optimizations are applied. We focus the evaluation on the effectiveness of checkpointing overhead reduction as measured by the following metrics: checkpoint buffer size reduction, the reduction in the number of `backup` calls, and the impact on the redundancy rate.

Checkpoint Buffer Size Reduction. Figure 11 shows the compiler optimization impact on checkpoint buffer size when all optimizations are incrementally applied. The effectiveness of our optimizations depends on the region size, as well as the frequency of stores within the region. Normally, a larger region has more opportunities for optimization. We observe that *RE1* is the most effective of all optimizations: as shown in Figures 11(a), and 11(b), respectively, *RE1*

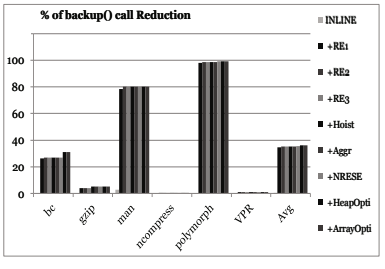


(a) large regions

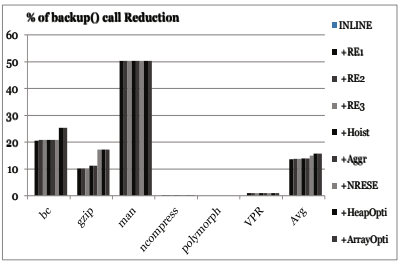


(b) medium regions

Fig. 11. Incremental/cumulative impact of optimizations on buffer size



(a) large regions



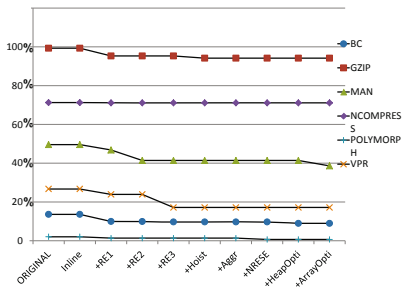
(b) medium regions

Fig. 12. Incremental/cumulative impact of optimizations on number of backup calls

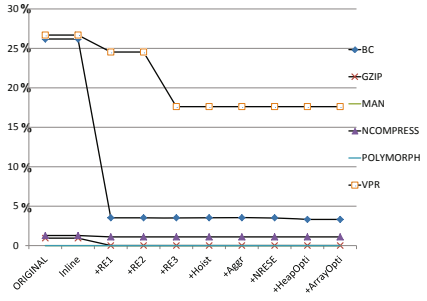
reduces the checkpoint buffer size by 92% in `polymorph`, and by almost 80% in `man`. When optimizations are incrementally applied, we observe a stable trend of buffer size reduction for both M and L regions. All performance numbers show that our compiler optimizations either exploit opportunities for optimization and hence improve checkpoint efficiency, or at least do not introduce negative effects (regressions). On average, the optimizations reduce checkpoint buffer size by an average of 52% for the L regions and 22% for the M regions.

Backup Call Reduction. in addition to buffer size reduction, our compiler optimizations also reduce the total number of `backup` calls—another metric for estimating the checkpointing overhead. Figure 12 shows that our optimizations reduce the total number of `backup` calls by an average of 36% for the L region and by 15% for the M region.

Redundancy Rate Impact. After all optimizations have been applied, it is interesting to understand how much redundancy remains in the checkpoint buffer, as a measure of what further optimization opportunities remain. We

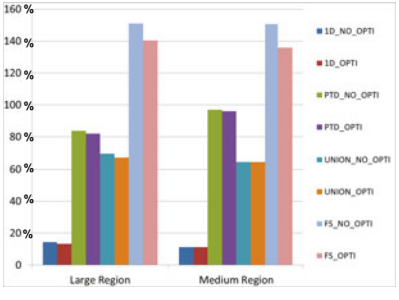


(a) large regions

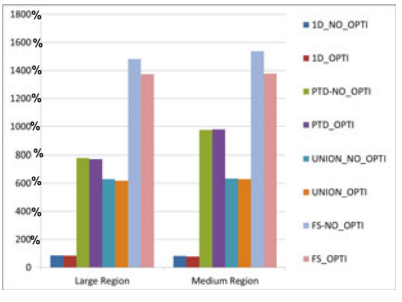


(b) medium regions

Fig. 13. Incremental/cumulative impact of optimizations on redundancy rate



(a) entire program



(b) within `try_swap` only

Fig. 14. Performance overhead of automated back-tracking support via our compiler-based checkpointing, relative to manually-implemented back-tracking support

quantify this by studying the region's *redundancy rate* (RR), as defined earlier in Section 4. Figure 13 illustrates the impact of our compiler optimizations on RR for both M and L regions when incrementally applying available optimizations. Figure 13(b) indicates that our optimizations are more effective in eliminating redundancy in M regions, since the highest RR is around 18% after optimizations. This is because the opportunities in M regions are more likely to be captured by our optimizations. Although optimizations do reduce redundancy for L regions as well, in most cases the impact is small. Three applications (`gzip`, `ncompress`, and `man`) still have very high RR even after applying all available optimizations. After manually examining the source code for each case we conclude that the high RR is due to extensive use of pointers, the presence of which hinders our optimization framework.

6.4 Overhead of Back-Tracking Support

In this section we evaluate the use of our checkpointing framework for implementing automatic back-tracking support in the VPR application, as introduced earlier in Section 5.2. Automatic back-tracking support frees the developer from having to manually implement support for checkpoint and restore, and also allows for source code that is easier to read and maintain.

We focus our evaluation on the `try_swap` function that implements the back-tracking portion of VPR. This routine spans almost 300 lines of source code including function calls and data access through link-list structures. Figure 13 shows that the final *RR* for VPR is 18%, and that *RE1* exploits most of the optimization opportunities. In addition to the 1D array buffer scheme, alternatively we can try a hash table solution to achieve a perfect redundancy behavior.[3] Section 4 introduces the details of our hash table designs, and their performance impact on checkpoint-enabled VPR is given in Figure 14.

Figure 14(a) presents the overall performance impact on VPR when our compiler-based automatic back-tracking is enabled. We observe that the 1D-array scheme achieves the best run-time performance after optimization. The entire VPR program has a mild 15% slowdown with 18% of buffer redundancy after applying all optimizations. When switching to buffers utilizing hash table schemes, we measure increased performance penalty to 62%, 95% and 140% when utilizing *union*, *PTD*, and *fixed* hash-table buffer schemes respectively.

We further zoom-in to the function level and measure the checkpointing performance impact on the `try_swap` routine alone (in Figure 14(b)) by comparing its execution time with and without our compiler-based checkpointing. Figure 14(b) is very similar to Figure 14(a). They can be treated as scaling on different levels of granularity. When measuring against `try_swap` function only, enabling automatic checkpointing will have performance penalties of 90%, 600%, 950% and 1500% for *1D array*, *union*, *PTD*, and *fixed* respectively.

7 Related Work

Our techniques leverage prior work in related areas, including support for software checkpointing, thread-level speculation (TLS), transactional memory (TM) and program back-tracking.

Checkpointing. Checkpointing is a process of taking program snapshots to facilitate later recovery. Feldman *et al.* [8] present the IGOR system capable of conducting full-process checkpointing, optimized for checkpointing only dirty pages. King's time-traveling VM [15] discusses an OS-level debugging facility by checkpointing entire OS states into disk files. Fine-grain refinement includes both undo-log and redo-log, to reach any specific program location between two consecutive full checkpoints. Xu *et al.* [32] demonstrate a re-tracing tool that

[3] Note that a hashtable always performs a search before any insert, thus the redundancy rate for all hashtable-based checkpoint buffer implementations is always 0.

uses VMWare's deterministic replay technique to collect only non-deterministic events during program execution and later expanding the collection into full program traces using replay. In contrast to existing approaches that checkpoint entire VM or application, we checkpoint on a per-store granularity to memory within a single application – a fine-grain checkpointing scheme that hasn't received much attention.

Speculation. Thread-level speculation [10,29] (TLS) and Transactional Memory [11,12,28] (TM) are optimistic program execution whose result might not be needed. TLS and TM approaches provide for each optimistic thread the ability to checkpoint and rollback, although this support is also intertwined with support for tracking and detecting inter-thread conflicts. Hardware buffering support for hardware TLS and TM implementations has the challenge that it can overflow. Software implementations can be less limited in buffer capacity, but suffer from high instrumentation overheads. In contrast to most TM or TLS solutions that using hardware buffering for multi-threaded workload, we instead focus on using software buffering for single-thread application. We further leverage compiler optimizations to aggressively reduce checkpointing overhead.

Program Back-Tracking. Debugging often requires revisiting passed program state while trying to locate the root cause of a bug. A checkpoint-enabled debugger can greatly simplify the debugging process by eliminating the need for program restart to look backwards. Agrawal *et. al.* [2,3] presented a prototype debugging tool that is based on dynamic program slicing and execution back-tracking—it provides a structured view of dynamic events through runtime traces, but is constrained by storage limitations. Recent versions of `gdb` [9] allow inverse execution by conducting program replay, but are limited to one million instructions. In contrast, our checkpointing scheme allocates its buffer in main memory so that it can grow dynamically. This allows a checkpoint region of relatively unbounded size and complexity. We expose the checkpointing functionality to the user, so that programmers can have explicit control of rewind by issuing debugger commands, to help reduce develop-run-debug cycle time and improve productivity.

8 Conclusion

We have designed, implemented and evaluated a comprehensive checkpointing framework that automatically enables software-only checkpointing over any user-specified source program region. In this paper, we presented compiler analyses and transformations that enable and optimize user-level checkpointing over programs of arbitrary size and complexity, and demonstrated that compiler optimizations are effective at eliminating checkpointing overhead. In particular, they reduce checkpoint buffer size by up to 98% and remove up to 95% of redundant `backup` calls. We showed that by leveraging our checkpointing framework, a debugger can conduct unlimited retries of execution rewind over arbitrarily large regions. We also showed that we can enable automatic back-tracking,

with a moderate performance overhead of only 15% for VPR's place-and-route algorithm.

Future Work. We plan to enhance our checkpointing API by allowing users to specify non-checkpointable code within a checkpoint region; this will have an immediate use for VPR because users will gain manual control within an otherwise automatically-checkpointed region. Redundancy rates remain high for a few applications after all optimizations due to extensive use of pointers, hence we plan to develop deep pointer analyses to better understand such pointer behaviors and help to further reduce checkpointing overhead. We also plan to extend our framework with multi-threading support, including evaluating a write-buffer approach.

References

1. Adl-Tabatabai, A., Lewis, B.T., Menon, V.S., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime optimizations for efficient software transactional memory. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2006)
2. Agrawal, H., Demillo, R., Spafford, E.: An execution-backtracking approach to debugging. IEEE Transactions on Software (May-June 1991)
3. Agrawal, H., Demillo, R., Spafford, E.: Debugging with dynamic slicing and backtracking. Software: Practice and Experience (October 2006)
4. Akkary, H., Rajwar, R., Srinivasan, S.: Checkpoint processing and recovery: An efficient, scalable alternative to reorder buffers. IEEE Computer Society (2003)
5. Betz, V., Rose, J.: Vpr: A new packing, placement and routing tool for fpga research. In: VPR: A New Packing, Placement and Routing Tool for FPGA Research (1997)
6. Betz, V., Rose, J., Marquardt, A.: Architecture and cad for deep-submicron fpgas. Kluwer Academic Publishers (February 1999)
7. Elnozahy, W., Johnson, D., Zwaenepoel, W.: The performance of consistent checkpointing. In: 11th Symposium on Reliable Distributed Systems, pp. 39-47 (October 1992)
8. Feldman, S.I., Brown, C.I.: Igor: A system for program debugging via reversible execution. In: ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging (1989)
9. Free Softwar Foundation. Gdb: the gnu debugger manual 7.0 (September 2009)
10. Hammond, L., Willey, M., Olukotun, K.: Data speculation support for a chip multiprocessor. In: ACM SIGOPS Operating Systems (December 1998)
11. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. In: CM SIGARCH Computer Architecture News (March 2004)
12. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N.: Software transactional memory for dynamic-sized data structures. In: The Twenty-Second Annual Symposium on Principles of Distributed Computing (2003)
13. Hwu, W., Patt, Y.: Checkpoint repair for out-of-order execution machines. In: Computer Science Division. ACM, University of California at Berkeley (1987)
14. Jagadish, H.V., Silberschatz, A., Sudarshan, S.: Recovering from main-memory lapses. In: Procs. of the International Conf. on Very Large Databases, VLDB (1993)

15. King, S.T., Dunlap, G.W., Chen, P.M.: Debugging operating systems with time-traveling virtual machines. In: Annual USENIX Technical Conference (2005)
16. Kingsley, G., Beck, M., Plank, J.: Compiler-assisted checkpoint optimization using suif. In: First SUIF Compiler Workshop (1995)
17. Lattner, C., Adve, V.: Llvm a compilation framework for lifelong program analysis and transformation. In: Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO) (March 2004)
18. Lattner, C., Adve, V.: The LLVM Compiler Framework and Infrastructure Tutorial. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 15–16. Springer, Heidelberg (2005)
19. Li, C., Stewart, E., Fuchs, W.: Compiler-assisted full checkpointing. Software-practice and Experience 24(10), 871–886 (1994)
20. Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., Zhou, Y.: Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the Evaluation of Software Defect Detection Tools (2005)
21. Mcdonald, A., Chung, J., Carlstrom, B.D., Minh, C.C., Chafi, H., Kozyrakis, C., Olukotun, K.: Architectural semantics for practical transactional memory. Computer Architecture News (2006)
22. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: Logtm: Log-based transactional memory. In: High-Performance Computer Architecture (2006)
23. Eliot, J., Moss, B.: Log-based recovery for nested transactions. In: Proceedings of the 13th International Conference on Very Large Data Bases (1987)
24. Ng, W., Chen, P.: The symmetric improvement of fault tolerance in the rio file cache. In: Proceedings of 1999 Fault Tolerance Computing, FTC (1999)
25. Plank, J., Beck, M., Kingsley, G.: Compiler-assisted memory exclusion for fast checkpointing. In: IEEE Technical Committee on Operating System and Application Environments, Special Issue on Fault-Tolerance (1995)
26. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent checkpointing under unix. In: Usenix Winter Technical Conference (1995)
27. Chandra, S.: An evaluation of recovery related properties of software faults. Ph.D. thesis (2004)
28. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L., Minh, C.C.: Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In: Principles and Practice of Parallel Programming, PPOPP (2006)
29. Gregory Steffan, J., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: International Symposium on Computer Architecture (ISCA) (June 2000)
30. Wang, Y., Huang, Y., Vo, K., Chung, P., Kintala, C.: Checkpointing and its applications. In: 25th Int. Symp. On Fault-Tol. Comp., pp. 22–31 (June 1995)
31. Whaley, J.: System checkpointing using reflection and program analysis
32. Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., Weissman, B.: Retrace: Collecting execution trace with virtual machine deterministic replay. In: 3rd Workshop on Modeling, Benchmarking and Simulation (2007)