Lightweight Snapshots and System-level Backtracking

Edouard Bugnion *Vitaly Chipounov* *George Candea*
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

We propose a new system-level abstraction, the lightweight immutable execution snapshot, which combines the immutable characteristics of checkpoints with the direct integration into the virtual memory subsystem of standard mutable address spaces. The abstraction can give arbitrary x86 programs and libraries system-level support for backtracking (akin to logic programming) and the ability to manipulate an entire address space as an immutable data structure (akin to functional programming). Our proposed implementation leverages modern x86 hardware-virtualization support.

1 Introduction

Operating systems and programming languages serve fundamentally different purposes: operating systems abstract hardware, manage resources, and provide applications with familiar abstractions such as address spaces, threads of control, and file descriptors. From an application perspective, the operating system ensures that the application can make forward progress by granting it resources and serving its system calls in an efficient and fair manner.

Programming languages, on the other hand, generally provide a level of abstraction that separates the representation of the computation—i.e., how the programmer expresses a problem—from its underlying execution—i.e., how the program gets executed on hardware. For example, logic programming languages such as Prolog enable the succinct description of exponential search problems by automating the backtracking process within the language runtime. Similarly, functional programming and its use of immutable data structures enables the natural decomposition of complex problems into tasks that can be executed in parallel [12].

Because they focus on different issues, commodity operating systems provide general-purpose abstractions and are generally oblivious to the specific requirements of programming language runtimes, and specifically when that runtime provides backtracking support as a first-order primitive.

We propose a new operating system abstraction: lightweight, immutable execution snapshots, which consist of a copy of the register file and an immutable logical copy of the entire address space of a process. They differ from a traditional address space abstraction because of their immutability. Unlike classic checkpoints [14], these snapshots are directly integrated into the virtual memory subsystem to enable the rapid creation (and destruction) of snapshot trees, and to initiate execution from any given snapshot. The snapshots are not scheduled by a traditional OS scheduler, but instead by one of the various well-understood search strategies, such as DFS, BFS or A$\star$, which provide a controlled exploration through a problem space. For example, we use the DFS scheduler to provide system-level support for fast backtracking of user-space programs. We can also use this snapshot abstraction to convert a program's address space into an immutable data structure of its own.

We argue that such an OS-level primitive can provide competitive performance for realistic problems by exploiting hardware virtualization, and propose an implementation sketch based on Dune [1]. Unique to our system, the exploration steps (the partial candidate extension step in backtracking terms) can be implemented in any language and runs as arbitrary x86 code without requiring any user-space bookkeeping. For example, a search problem can be written in any programming language as a simple "single path to solution" program without having to worry about undoing any side-effects. Instead, it simply relies on the system software to guess (or appear to guess) each decision along the path.

By moving backtracking and lightweight, immutable data structures from the field of programming languages and runtime libraries into the operating system space, we see opportunities to broaden the scope of solutions that can benefit from the approach, for example with many applications that currently rely on ad-hoc mechanisms to emulate snapshots or backtracking operations.

2 The Need for Speed

Our interest in the systematic, controlled exploration of an exponential search space was triggered by the rising (practical) use of symbolic execution in testing and verification [6, 9]. In such an approach, some (hopefully many, ideally all) possible paths of a program or system are explored systematically to validate or invalidate assertions about a particular program [6] or an entire virtual machine [9]. We show here two different applications that could benefit from lightweight snapshots and system support for backtracking:

Testing and verification of program binaries. S^2E [9] is a platform for writing tools that analyze the properties and behavior of software systems. So far, we have used S^2E to develop a comprehensive performance profiler, a reverse engineering tool for proprietary device drivers [8], a bug finding tool [13], and a tester for file system code [7]; others have used S^2E for a number of other tools [15, 18, 17]. The platform enables developers to simultaneously analyze entire families of execution paths, instead of just one execution at a time, and to perform these analyses in-vivo within a real and complete software stack of a virtual machine.

S^2E combines symbolic execution based on KLEE [6] with the QEMU virtual machine [2]. S^2E employs backtracking when exploring multiple paths of execution of a virtual machine that has a combination of symbolic and concrete inputs. Conceptually, S^2E is an automated path explorer with modular path analyzers: the explorer drives the target system down all execution paths of interest, while analyzers check properties of each such path (e.g., to look for bugs) or simply collect information (e.g., count page faults). When searching for bugs, e.g., one may direct the S^2E explorer down the paths that are likely to have such bugs and let analyzers check whether the desired properties hold.

At the core of S^2E exploration is a *conceptual fork* of the entire state of the VM. This is currently implemented by snapshotting in software all QEMU data structures and the VM. The snapshot is optimized by emulating copy-on-write behavior within QEMU itself. Even though S^2E can scale to large systems, such as a full Windows stack, it faces significant inefficiencies resulting from the fact that multiple (relatively fat) software layers need to be "tricked" into doing the right thing to implement copy-on-write of symbolic system state.

System-level hardware-assisted backtracking would dramatically cut the implementation complexity of S^2E and increase performance. S^2E currently modifies about 2 KLOC spread in the QEMU's code base (about 800 KLOC) in order to catch all register and memory writes. These changes implement copy-on-write and ensure that accesses to symbolic data call the S^2E emulator. System-level backtracking can remove all the ad-hoc instrumentation and cut several layers of indirection, including the software MMU emulation. The resulting performance gain would allow S^2E to verify larger software, find more bugs, and achieve higher code coverage faster.

SAT/SMT solving. Finding values for variables of a given Boolean formula that make the formula evaluate to true (i.e., the SAT problem) is a fundamental goal for much of computing. SAT's younger cousin, SMT ("satisfiability modulo theories") is a similar decision problem, but for logical formulas in classical first-order logic with equality; examples of theories under which SMT can be formulated include the theory of real numbers, of integers, of lists, arrays, bit vectors, etc. SAT and SMT solvers are extensively used in software and hardware verification, constraint solving in artificial intelligence, operations research, electronic design automation, and many other areas. Both SAT and SMT are NP-complete problems (SMT can even be undecidable in certain situations). As a result, solvers generally implement either a systematic backtracking search procedure to explore the (exponentially sized) space of variable assignments looking for satisfying assignments, or take a randomized heuristic approach. In essence, they are one of the quintessential users of backtracking.

While our proposal cannot magically turn NP into P, it can help make solvers faster. For example, modern SMT solvers (like Z3 [10]) can reduce the time it takes to find a satisfying assignment by leveraging the intermediate data structures and results of previously solved constraints. Specifically, an *incremental* solver given formula p immediately followed by formula $p \wedge q$ can solve both in less time than solving p and then solving $p \wedge q$ from scratch without leveraging the knowledge of p. By creating a lightweight snapshot for solved problem p, we can ensure that $p \wedge q$ is solved incrementally.

3 System-level backtracking

Logic programming allows applications to seemingly execute multiple paths through a search space in a determined order, while providing the simple programming model of executing through a single path. System-level backtracking aims to provide the same illusion though operating system primitives. Figure 1 uses the classic n-queens problem to illustrate how to use system-level backtracking: first, the `main` function selects `DFS` as a search strategy; then, for every column, the `sys_guess` system call returns an index between 0 and $N-1$ and provides the user-space program with the illusion that the operating system has *guessed* the path to the solution; extension steps backtrack using the sys_guess_fail system call, similar to the use of fail in Prolog. Once a puzzle is completed, the answer is printed to stdout. As in Prolog, we can simply use backtracking to print all answers to the puzzle. We note that the implementation *appears* to execute in linear time, and does not require any manual instructions to *undo* changes to the state.

A naive implementation of sys_guess and sys_guess_fail would simply use the POSIX fork, wait and exit system calls. Sequential depth-first-search (DFS) exploration of a search problem could be implemented by simply issuing a fork before exploring any extension off that partial candidate, and having the child process explore the subtree while the parent waits for completion. A parallel depth-first-search strategy might simply fork without waiting, with possibly dire consequences.

However, using fork as the foundation for system-level backtracking is inappropriate for a number of reasons. First, fork creates both a new address space and a new thread of control. Although the former is required to ensure isolated execution, the latter is undesirable. Instead, search algorithms require the systematic, controlled exploration of the problem space. Second, forked processes are neither isolated from each other nor encapsulated, e.g., shared file descriptors provide problematic communication channels and changes made to files are visible to other processes. And last but not least, the large performance overheads of this naive approach would likely dwarf any benefit in most circumstances.

3.1 Concepts and Abstractions

Our proposed system-level abstractions use the classic backtracking terminology: a *partial candidate* is an immutable state abstraction, and collectively, the partial candidates form the vertices of the search graph; a *candidate extension step* is a deferred computation abstraction that, when evaluated against its parent partial candidate, can generate a new partial candidate and new extensions. The extensions form the directed edges of the search graph. Finally, the algorithm is controlled by a *search strategy* (such as DFS), which schedules the evaluation of extensions.

We apply backtracking to the system-level abstraction of a single-threaded process:

Partial candidates. A partial candidate is a state abstraction, which consists of the combination of an immutable register file, an immutable address space, and immutable files. Each sys_guess system call creates a new partial candidate that is the lightweight immutable snapshot of the currently executing thread. Each partial candidate also has an immutable relationship with its parent, which can be leveraged to encode the state in a space-efficient manner.

```
void nqueens(int N)
{
  for (int c=0;c<N;c++) {
    int r = sys_guess(N);   // a little magic;
    if (row[r]||ld[r+c]||rd[N+r-c])
      sys_guess_fail();           // backtrack;
    col[c] = r;
    row[r] = c+1;
    ld[r+c] = 1;
    rd[N+r-c]= 1;
  }
  printboard(N);
}
main() {
  if (sys_guess_strategy(DFS)) {
    nqueens(8);
    sys_guess_fail();    // print all answers;
  }
}
```

Figure 1: N-queens with system-level backtracking.

Candidate extension steps. A candidate extension step consists of the execution of arbitrary x86 code in a controlled environment: the starting point is the combination of a lightweight snapshot with a return value from sys_guess corresponding to the extension number. Candidate extension steps subsequently execute in an isolated fashion to not violate the immutability of the parent partial candidate, and to not accidentally communicate with other extension steps currently executing, or any external entity. This implies that all system calls issued by the extension step are appropriately interposed on.

Failure. Reaching a contradiction is intrinsic to backtracking problems. The sys_guess_fail system call, similar to Prolog's fail, simply discards the currently executing extension steps and never returns.

Flexible search strategies. The search strategy is implemented separately from the extensions or the partial candidates. It implements a policy that schedules the next extension to be evaluated on a given thread. This includes classic search strategies such as DFS, BFS and A$\star$. These are all internally driven strategies where the search exploration process generates a stream of candidate extension steps to be evaluated. In addition, we can support externally controlled search strategies where an external entity can generate new extension steps for any given partial candidates, and schedule their execution.

New system calls. As extensions run arbitrary x86 code, they communicate through system calls that map directly to the backtracking framework. The three system calls of Figure 1 provide the minimal API required for simple search strategies such as DFS and BFS. In addition, search strategies that rely on goal-distance heuristics such as A⋆ and SM-A⋆ require that the distance vector of the extension steps be communicated via an extended `guess` system call. Additional APIs can be envisioned to allow a richer interaction with the system, e.g., to selectively encapsulate I/O interactions, control execution timeouts, or create explicit sharing mechanisms between lightweight snapshots.

3.2 Applications

Our two motivating examples map to the concepts as follows: in S^{2}E, each partial candidate corresponds to a different state of the VM (consisting of the concrete state augmented with symbolic data and symbolic constraints), executed up to the point where a symbolic branch condition is encountered, i.e., a branch whose condition depends on symbolic values. The evaluation of an extension is the simulation of the virtual machine execution (by QEMU and KLEE) until it terminates or reaches the next symbolic branch. At that point, it creates a partial candidate together with the two extensions corresponding to the branch *taken* and branch *not-taken* constraints. S^{2}E uses a search strategy (such as DFS or a coverage-optimized strategy) to select partial candidates according to problem-specific heuristics.

With incremental solvers, partial candidates correspond to solved SAT/SMT problems, complete with their intermediate data; extensions represent incremental clauses that are logically combined with the parent partial candidate clauses. When incremental solvers are used by symbolic execution systems, they typically run as a library loaded within the address space of the symbolic execution engine. Then, the natural positive side-effect is that the incremental solver will be able to build upon the prior solved problem since it is its parent's.

Alternatively, one could use lightweight snapshots directly to create a multi-path incremental SAT/SMT solver *service*, built using a single-path incremental solver. In this case, the service waits for client requests consisting of of an opaque reference to a previously solved problem p and an incremental constraint q, and returns to the client the solution to $p \wedge q$ together with a opaque reference to that new problem.

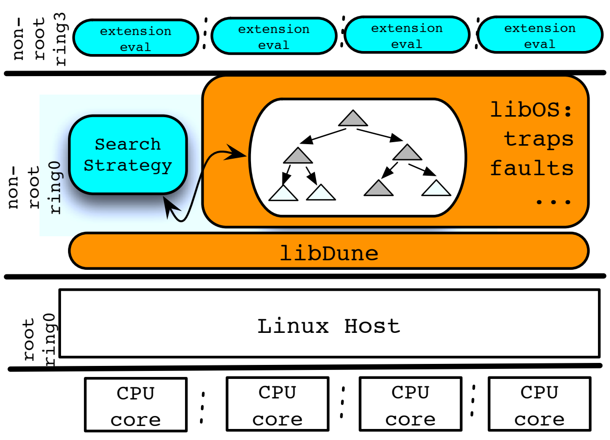

Figure 2: System architecture. Filled triangles represent partial candidates; arrows represent extensions.

4 Implementation Sketch

Although we here claim that adding system-level backtracking support can help with a number of relevant applications, the community is also aware of the difficulty of getting any operating system modifications into the mainstream. Fortunately, we are building our lightweight snapshots and system-level backtracking support directly on top of the Dune framework [1], which loads as a standard Linux kernel module and leverages hardware virtualization to safely expose hardware features to library operating systems (*libOS*). However, and unlike prior-generation libOS-es that were designed to run on top of exokernels [11] or virtual machine monitors [5], Dune enables our libOS to run as an application on top of an unmodified commodity OS like Linux.

Dune directly takes advantage of two hardware features introduced in x86 CPUs to support virtual machines: (i) VT-x [19] (or AMD-v), which supports CPU virtualization and (ii) nested page tables [3]. The former enables the creation of a protected libOS, which can intercept all system calls, whereas the latter enables the libOS to directly create and manipulate address spaces and efficiently handle page faults. The original evaluation of Dune [1] is promising, showing for example that memory protection events and forks can be implemented via a specialized libOS with an order of magnitude better performance than corresponding Linux abstractions.

Figure 2 describes the architectural building blocks involved in our proposed implementation. Lightweight snapshots and system-level support for backtracking are implemented as a Dune libOS running at `ring 0 (non-root)`. The libOS builds on the Dune sandbox application, in particular to load the application in `ring 3`. From the perspective of the host OS, the libOS runs as a single multi-threaded process, with the number of threads typically corresponding to the number of hardware threads. The libOS manages the internal structures of the search graph: each partial candidate is a lightweight immutable snapshot consisting of the register file, a logical copy of the guest-virtual address space accessible to applications, and a logical copy of open disk files; unevaluated extensions are simply a reference to their parent partial candidate and the extension number; the libOS's scheduler selects the next unevaluated extension, restores the lightweight snapshot, sets the extension number into `%rax`, and resumes execution at `ring 3`.

The libOS is in charge of handling the page faults and system calls resulting from the evaluation of candidate extensions. Unique to our system, these extensions run as arbitrary x86 code that can make arbitrary system calls—the libOS interposes on these calls to ensure that all visible side effects are contained within the extension. Page faults are dominated by the copy-on-write faults that guarantee the immutability of the parent snapshot.

5 Discussion

We implemented a proof-of-concept prototype as a Dune libOS, which supports DFS, BFS, and A$\star$ search strategies, and can run complex software such as Z3 [10]. It is not yet optimized, has only partial support for system-call interposition, and supports only single-threaded execution. When applied to toy applications like n-queens, our prototype performs (as expected) substantially worse than a hand-coded implementation, but better than a Prolog implementation running on XSB [16].

Problem granularity and memory locality. One of the primary design goals is to minimize the overheads of system-level backtracking, for example as compared to a native implementation that hand-codes the backtracking or state forking logic. Clearly, problems with a trivial instruction count per extension step (e.g., n-queens) are best implemented by hand-coding the backtracking logic on a stack. But our motivating examples have address spaces measured in GB, the software that performs the extension evaluation consists of many thousands of lines of code and touches dozens or even hundreds of 4-KB pages during a single extension step. The execution granularity, complexity of hand-coded logic, and page-level memory locality will each play a role to determine when the approach provides a performance win.

Immutable data structures. Functional programming revolves around the manipulation of immutable data structures, which simplify both sequential and concurrent programs. Although imperative languages have existing libraries that efficiently implement immutable sets, maps, and other simple structures, domain-specific immutable types can be more difficult to write. Lightweight snapshots provide a very coarse, yet very simple to use, immutable type: the entire address space of the program. Our motivating example of incremental solvers is only one of many examples of that paradigm.

System call interposition. The framework intercepts system calls to ensure the isolated execution of the extension. At the very least, any call that changes the address space (e.g., `brk`) must be logged and reversed upon backtracking. This interposition logic can easily be made sound by supporting only the minimal required set of conditions (e.g., only open regular files but not devices) and failing all others. Making the interposition logic complete does not appear tractable (e.g., the challenges in interposing on socket I/O with a remote peer).

6 Related Work

Our approach builds on the Dune framework and its sandbox libOS. Like Dune, we leverage virtualization hardware to create new process-level abstractions.

Lightweight, immutable snapshots are a form of checkpointing [14]. However, our approach differs in that we establish the snapshot as a system-level abstraction, fully integrated with the virtual memory subsystem of our libOS, and designed to both take and restore snapshots with very high frequency. Wedge [4] (as implemented in Dune) has a similar integration, but for the different purpose of thread recycling.

Our approach bears some similarity to the Warren Abstract Machine (WAM) [20], a way of implementing interpreters for Prolog, with our `sys_guess` calls corresponding to the WAM choice points. Our solution is unique in that it operates exclusively with hardware-defined concepts such as the register file and the paged virtual address space, making the backtracking logic substantially simpler and more efficient.

7 Conclusion

We have presented a new system-level abstraction, the lightweight snapshot, which directly integrates into the virtual memory subsystem of a libOS relying on hardware virtualization support. We made the case for adding system-level support for backtracking through the combination of lightweight snapshots, system call interposition, and a minimal set of new system calls. Our approach uniquely allows arbitrary x86 programs to perform partial candidate step extensions without any backtracking bookkeeping done by the program.

References

[1] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazieres, and C. Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Symp. on Operating Sys. Design and Implem.*, 2012.

[2] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conf.*, 2005.

[3] Ravi Bhargava, Ben Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[4] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Symp. on Networked Systems Design and Implem.*, 2008.

[5] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[7] João C. M. Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. Scalable testing of file system checkers. In *ACM EuroSys European Conf. on Computer Systems*, 2012.

[8] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. In *ACM EuroSys European Conf. on Computer Systems*, 2010.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[10] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[11] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Symp. on Operating Systems Principles*, 1995.

[12] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[13] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conf.*, 2010.

[14] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under UNIX. In *USENIX Annual Technical Conf.*, 1995.

[15] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. Symdrive: Testing drivers without devices. In *Symp. on Operating Sys. Design and Implem.*, 2012.

[16] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB as a deductive database. In *ACM SIGMOD Conf.*, 1994.

[17] Raimondas Sasnauskas, Oscar Soria Dustmann, Benjamin Lucien Kaminski, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. Scalable symbolic execution of distributed systems. In *Intl. Conf. on Distributed Computing Systems*, 2011.

[18] Raimondas Sasnauskas, Philipp Kaiser, Russ Lucas Jukić, and Klaus Wehrle. Integration testing of protocol implementations using symbolic distributed execution. In *Intl. Workshop on Rigorous Protocol Engineering*, 2012.

[19] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, 2005.

[20] David H. D. Warren. An abstract prolog instruction set. Technical report, SRI International, October 1983. Technical Note 309.