Trials and Tribulations in Synthesizing Operating Systems

Jingmei Hu
Harvard University
Cambridge, MA, USA

Eric Lu
Harvard University
Cambridge, MA, USA

David A. Holland
Harvard University
Cambridge, MA, USA

Ming Kawaguchi
Harvard University
Cambridge, MA, USA

Stephen Chong
Harvard University
Cambridge, MA, USA

Margo I. Seltzer
University of British Columbia
Vancouver, BC, Canada

Abstract

Recent advances in program synthesis convinced us that it was the right time to transform the process of porting an operating system into a program synthesis problem. We set out to synthesize the needed machine dependent code for an existing operating system. This undertaking proved far more challenging than we anticipated. We summarize our experience and lessons learned and propose next steps in realizing such an undertaking.

1 Introduction

Porting operating systems is tedious. For much of the past two decades, we've been able to ignore this problem: everyone ran x86 or x86-compatible hardware and the only porting we did was upgrading to new versions of the x86 architecture. The end of Moore's Law changed all this. Today we have different kinds of CPUs (ARM, x86, RISC-V), special purpose devices (GPUs, TPUs, DSPs), and customizable devices, such as FPGAs and ASICs. Or, in the words of Hennessy and Patterson, we've entered the "New Golden Age for Computer Architecture" [11]. Many of these new devices require systems software, returning us to the business of porting operating systems.

In parallel, program synthesis and verification have become more practical. The convergence of hardware diversity and breakthroughs in program synthesis convinced us that it was time to synthesize the machine dependent parts of an operating system. We decided to start with an existing operating system; this was our first mistake. In hindsight, we should have written something tiny on which to experiment. We will discuss the ramifications of this choice in section 3. We decided to use Barrelfish [2], because its decomposition into a CPU driver and user-level monitor seemed ideal in its separation of machine dependent and machine independent code. Unfortunately, this meant that we were working with completely unfamiliar system. (However, the Barrelfish team was incredibly helpful and accessible.)

On the program synthesis side, we initially decided to start with the Rosette [28] symbolic virtual machine to develop our domain specific languages for specifying operating system functionality and describing machine architectures. Rosette is a metalanguage that allows a user to design *solver-aided* DSLs. Implementing a DSL under Rosette allows quick application of synthesis algorithms (such as CEGIS) by using an SMT (Satisfiability Modulo Theories) solver under the covers (e.g. Z3 [6]), without requiring the DSL implementation to formulate SMT queries or handle solver feedback such as raw counterexamples. In particular, our initial implementation generated Rosette code from a machine description. Then, it used Rosette to do symbolic execution of the assembly language and find an assembly program satisfying a specification.

Rosette worked well for small synthesis problems: those doing only pure register manipulation with no memory access and up to three or four instructions. As we increased the scope of our synthesis targets, we ran into efficiency problems: some of our doing and some inherent to assembly synthesis. Pointer and memory handling turned out to be an important source of slowness. We wanted to explore different approaches to symbolic execution and memory handling, but found that doing so with Rosette would need substantial engineering across Rosette's symbolic execution abstraction layer. Ultimately, we chose to write our own symbolic execution engine in OCaml.

This engine is part of our synthesis engine, which is in turn part of an ecosystem of languages and code generation tools. For perspective, consider what might be involved in creating a single kernel source file. Figure 1 shows a (simplified) diagram. We generate code as basic blocks in assembler;

PLOS'19, October 27, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-7017-2/19/10…$15.00
https://doi.org/10.1145/3365137.3365401

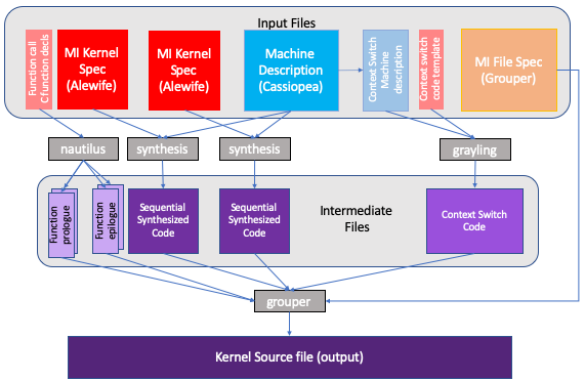

Figure 1. Sample synthesis data flow

```
require value wordsize: int
require value ret: wordsize reg
require value wordzero: wordsize bitvec
require function interrupts_are_on: () bool
block cpu_irqon {
        pre: interrupts_are_on() == false
        post: interrupts_are_on() == true }
block setret_zero {
        pre: true
        post: *ret == wordzero }
```

Figure 2. Two Alewife specifications: to enable interrupts, and to set the return value register to zero.

we have multiple code generation tools. Grayling is a special-purpose compiler for generating context switch code; Nautilus generates calling sequences and function prologues and epilogues. Synthesis proceeds from machine-independent specifications written in our specification DSL Alewife and a machine description written in our description language Cassiopea[1] [12]. A block composition tool, Grouper, collects the blocks and produces the final output. Our synthesis engine can synthesize small components from preexisting operating systems for multiple architecture platforms.

Figure 2 shows two small Alewife specification blocks, which enable interrupts and assign zero to the return value register, respectively. Alewife declares pre- and postconditions for a block of assembly code as well as abstract states and functions which are then concretely defined in Cassiopea for each machine.

Our goal in writing this paper is to expose the challenges that arise in synthesizing operating system code, share our experience, and propose future directions. Although our undertaking was overly ambitious, we continue to believe that the future of OS portability lies in program synthesis.

In the next section, we give brief overviews of modern program synthesis and of synthesis in operating systems. In section 3 we discuss our own blunders in a bit more detail before moving to section 4, which outlines the fundamental challenges in OS synthesis. Section 5 suggests ways forward, and we conclude in section 6.

[1]Cassiopea is named after a jelly(fish), not the mythological queen, and is spelled accordingly.

2 Background

2.1 Modern Program Synthesis

Modern synthesis dates from 2006 with SKETCH [26]. A "sketch" is a partial program with textual "holes" in it that must be replaced with program elements. SKETCH and its descendents combine this partial program with a specification of the missing functionality and search the space of possible programs for an implementation, using automated verification to check for correctness. If a candidate program satisfies the specification, synthesis is complete; if not, the verification procedure produces a counterexample to factor into finding the next candidate. This approach evolved into "counter-example guided inductive synthesis" or CEGIS [25]. We adopted the CEGIS algorithm for synthesizing small blocks of machine-dependent operating systems code; we construct full kernel source files from collections of code blocks using a technique resembling sketching.

We did not adopt iterative techniques that require user interaction during synthesis, such as iterative refinement [3] or input/output synthesis (e.g., programming-by-example), as they seemed poorly matched for generating assembly code [8, 10]. Although we are not currently using hybrid techniques that combine multiple synthesis approaches, we have begun experimenting with deductive synthesis and have reason to believe that incorporating deductive synthesis will be useful. However, existing techniques, such as FlashMeta [19], which combine inductive synthesis with input/output examples, do not seem to be quite appropriate. Instead, we intend to use similar deductive techniques to constrain CEGIS.

Other approaches to constraining and ordering the search space, such as Feng et al.'s Neo [7], which combines sketching-style CEGIS with deep learning, derived from Balog et al.'s DEEPCODER [1], could prove helpful.

Another approach, reactive synthesis [16], allows for the synthesis of code that responds to changes in a finite input state by updating a derived output state according to a specification. This also does not fit our environment very well; in particular the processor state we interact with is not an independent automaton.

2.2 Synthesis in Operating Systems

Reactive synthesis is the methodology behind the Termite/Termite2 device driver synthesis project [22, 23], which is the main prior application of synthesis to operating system code. Device drivers pose a problem rather different from the one we address. The synthesis problem for drivers is framed as the driver code reacting to changes triggered by either the OS or the hardware device. Most of our synthesis problems

have one of two forms: either update the CPU state in a particular way when triggered by the OS or vice versa; in both cases what needs to be synthesized is not the choice of update but the specific way to get it done at the machine level, which Termite2 does not handle. (Termite2 is thus almost completely complementary to our work.)

There is a long history of generating code to marshal and unmarshal messages [21, 24]. However, this form of code generation bears little resemblance to modern synthesis; it is more accurately described as compiling an interface description language. Similarly, Massalin's Synthesis kernel [20] and the Scout operating system [14] use runtime code generation to create specialized code for fast-path execution. These efforts are also unlike modern synthesis in that they specialize preexisting general purpose code into streamlined versions of particular paths through the kernel or network stack.

3 Our Mistakes

We divide the challenges that arose in this project into two categories: those self-inflicted and those fundamental to the task. We include the problems we brought onto ourselves to help others avoid making the same mistakes.

As mentioned above, we chose to focus on porting a real, fully-functional operating system. Our initial goal was to be able to run a large, complex Java application on the synthesized operating system. Thus, we selected our system based on its ability to run a JVM. While courageous, this was foolish. We greatly underestimated just how difficult it was going to be to synthesize the tiniest pieces of an operating system, let alone a sufficiently complete machine dependent layer to run a JVM. We should have selected a few small pieces of functionality from the complex application, developed them as standalone C programs, and then selected a system that could feasibly run them. The fallout from this mistake was significant. We invested an enormous amount of time learning a large, new system. We wasted time researching JVMs — which ones might run on Barrelfish, which ones were sufficient to run the application, etc. None of this furthered our real research goal.

More fundamentally, there was evidence from past projects that undertaking such an audacious project would have been significantly more feasible if using a system designed for the task. Massalin's Synthesis kernel [20], which used runtime code generation (compilation, not synthesis in the modern sense), to produce fast-path custom implementations of common behaviors, is a great example. They did not try to add runtime code generation to an existing kernel, but instead developed a kernel specifically designed for the task. In retrospect, a kernel designed for synthesis would have a much cleaner API between machine dependent and machine independent code and it could be built to minimize the amount of assembly code (see subsection 4.3).

Lesson 1: One miracle per program[2]*. In other words, focus on the core research contribution to avoid being seduced into grandiose ambitions.*

The second big mistake we made was not modeling all data as bitvectors. All machines use machine integers of some fixed size, which in the SMT solver can be translated into bitvectors. However, solvers also support reasoning about mathematical integers. It seemed attractive to treat some values as bitvectors, such as bits within registers (e.g., when setting/clearing the interrupt bit in a status word), and other values as integers, such as the memory offset index in a load or store instruction. Although most SMT solvers provide conversions from integer to bitvector and back, reasoning about the conversion is time-consuming. Mixing bitvectors and integers during synthesis leads to poor performance in the SMT solver. Furthermore, some SMT solvers, such as Boolector [5], do not provide integer arithmetic support. Early on, we convinced ourselves to treat some values as integers; then we encountered the overhead in conversion and the resulting performance problems. We probably would have benefited greatly from early experimentation that revealed how much mixed representation cost us.

Lesson 2: Experiment early and often. When making fundamental design and/or modeling decisions, gather data on the downstream effects.

The third mistake was trying to use assembly language for everything. We felt that since we needed assembly language for some things that we might as well use it for everything. That way we would not have to develop two of everything. We significantly underestimated the difficulties of handling assembly language relative to imperative higher-level languages. This cost us when we were ready to move beyond toy examples.

Lesson 3: It is good to keep things simple; but it is not so good to put all your eggs in one basket in the process.

4 Fundamental Challenges

There are two aspects of OS synthesis that make the problem more challenging than other synthesis tasks: some tasks are fundamentally difficult to specify, and for others, the needed output is large leading to scalability issues. Synthesizing assembly, in particular, greatly exacerbates scalability challenges (subsection 4.3). Naturally, some parts of the operating system exhibit both challenges.[3]

4.1 Specification Challenges

Conventional program synthesis tends to produce programs that compute things, e.g., formulas, functions over data structures, or numerical methods. However, operating systems

[2]This idea is not original to us, but we failed to find an original citation for it.

[3] Thanks to James Bornholt for articulating this so clearly for us.

do not compute things; the machine dependent parts especially do not compute things. Instead, they change state. While some of these state changes are easy to specify, such as turning interrupts on, some are difficult to specify in a machine-independent manner, such as initializing the processor at boot time. And others are more challenging outright, such as flushing the instruction cache. Meanwhile, the need to model machines at the register level means our machine description language is register transfer language (RTL) style, which itself makes some of these problems more difficult.

The difficulty and complexity of initializing the processor at boot time varies widely from architecture to architecture. So does the amount and nature (and even the location) of the machine state that must be adjusted. The worst case is probably the x86, where the complete initialization sequence involves not only transiting through multiple processor modes that do not even use the same instruction sets, but also initializing a variety of in-memory data structures used by the processor at runtime for entirely machine-dependent purposes. (This is in addition to other state that reflects machine-independent concepts, e.g. caches and page tables.) No single aspect of this initialization is overly hard to specify as long as the specs are machine-dependent so they can refer to the machine-dependent concepts. However, it is difficult to write anything machine-independent about it.

However, it gets worse. While RAM caches are perhaps less hands-on today than in the past, kernels must still manipulate them at times, such as when reading code into memory. This requires specifications for both the available cache control instructions and the kernel's cache-control interface, so that the correct cache lines are selected and the correct actions taken upon them. Unfortunately, cache state in modern CPUs is complicated and varies even between models of the same architecture. Often the details are not publicly available. Describing the exact cache hierarchy is not feasible; and even if it were, the resulting model would be too large for a synthesis engine to process in a reasonable length of time. Meanwhile an RTL-style description gives little facility for abstraction; existing work on modeling caches focuses on timing behavior [29] and on coherence protocols [13], not on management and control.

Specification of the behavior of timers and timer registers is also difficult, especially in RTL. Today most CPUs have on-chip timers and OSes want to use them for time-slicing. Such a timer might work by having a register that decrements at the CPU clock rate, generating an interrupt when the register reaches zero. Alternately, some architectures implement timers with one register that increments at the CPU clock rate and another that specifies a value at which the timer should generate an interrupt. A machine-independent specification must account for both of these, and probably various other idioms. This introduces myriad problems. First, in RTL, it is difficult to handle registers that update spontaneously. Second, one must be able to relate the tick rate to external time units. Third, one has to figure out how to specify that the value to be loaded into a register *is* a tick count so that the synthesizer can select an appropriate register. And fourth, one also has to be able to specify what it means to trigger an interrupt, although this burden can perhaps be shifted to machine-independent code. Is it necessary to include temporal logic in the synthesis framework in order to be able to handle timers? This remains unclear.

Lesson 4: There is a reason some things are in the machine dependent layer; they are, sometimes, not only machine dependent, but machine specific.

A second problem related to specification is that for some things, the amount of effort required to specify and synthesize behavior far exceeds the amount of effort needed to write the code by hand, leading to a truly awful power-to-weight ratio. The clearest example of this is, perhaps, memory barrier instructions. Concurrent memory models are difficult to specify [4], and using those models to specify the behavior desired by an OS is difficult as well. Code synthesis over these models may also be problematic. But in all cases the output of synthesis is *a single instruction*, one from a set of at most a small handful offered by the architecture. Writing down the instruction by hand is both easier and faster. One saving grace is that the modeling gives some formal assurance that the instruction selected is the *correct* one out of the possible memory barriers; but even this is only as good as the specification of the behavior expected by the OS. The cost of the OS specification can be amortized over many ports, but even then, each processor's memory model must still be converted to a compatible formalism.

Lesson 5: If you are going to write specifications that are far longer and more difficult to write than the synthesized code, you really need a lot of different implementations that can leverage those specifications.

4.2 Scalability Challenges

While some things are difficult to specify, other are conceptually easy to specify, but push the scalability limits of synthesis tools. For example, a context switch is conceptually simple: machine state is copied to some location in memory and state in another location of memory is loaded into the machine. Many architectures have 32 registers; generally on a trap nearly all of these need to be saved, and usually some additional special-purpose or control registers will need to be saved too. On most architectures each one of these takes at least one instruction and some will take two (or more), so there will be on the order of 40 instructions just to do the save. An assembly trap handler needs to do other things as well. For example, MIPS has 32 registers, and on NetBSD 9 the general trap handler[4] varies considerably with kernel build options but requires approximately 60 instructions for trap

[4] `kern_gen_exception` for mips3 and up.

entry. This is close to an order of magnitude more instructions than we've been able to synthesize to date. We ended up abandoning synthesis for context switches and instead wrote a compiler (Grayling, mentioned above) specifically for the bulk register moves found in context switch code. It uses machine-independent register group specifications and machine-dependent lists of registers to generate both the load and save assembly blocks and the C data structures (trap frames, `jmp_buf`, etc.) upon which they operate.

Lesson 6: Some components of the machine dependent operating system are not amenable to program synthesis techniques; consider a hybrid approach before devoting significant time to synthesizing them.

In section 1, we mentioned one reason that we ultimately abandoned Rosette. Besides exploring other approaches to symbolic execution, we also hoped to gain additional transparency for the purposes of identifying scalability bottlenecks. Mixing integer and bitvector arithmetic resulted in poor performance on the part of the Z3 solver, which Rosette uses under the covers to implement CEGIS. In particular, Nelson-Oppen theory [15] combination, the algorithm used by Z3 for combining theories, only allows equality constraint propagation between theories, in this case linear arithmetic and finite bitvectors. We gave the SMT solver queries that combined the two theories in a way that required sharing arithmetic constraints, which is algorithmically impossible in current solvers. Instead, Z3 was forced to use an inefficient implementation of bitwise integer arithmetic, often called "bit-blasting", that resulted in solver performance an order of magnitude slower than expected. However, because Z3 is used as a library by Rosette, we could not initially tell which of Rosette, Z3, or both, was the source of our performance bug, nor was it immediately clear how to fix the bug.

Lesson 7: Despite recent advances in efficient SMT solving, tools that use solvers under the covers can still yield unpredictable behavior, particularly when misused.

Corollary 7: Examine exactly what solver queries you are generating and the performance of your solver on them. The results may be surprising (and problematic).

4.3 Assembly Language is Hard to Synthesize

Synthesizing assembly code creates unique scalability challenges when using symbolic execution and CEGIS. In particular, assembly languages generally have the following characteristics that differ from higher-level languages: First, all state is global. Second, all data is untyped and unstructured. Third, some arguments are drawn from exceptionally large spaces, such as bitvectors, which are exponential in the number of bits. Finally, the space of programs is exceptionally large, even when the program size is small. Even worse, because all state is global and data untyped, the input state to any instruction depends on all preceding instructions. In contrast, higher-level languages typically limit accessible state using scoping and data encapsulation (such as objects), have strong datatypes that restrict the space of arguments to an operator or function, and restrict the number of operators syntactically within the language.

In combination, these characteristics of assembly languages cause a combinatorial explosion in the number of possible instructions that must be considered. That is, the space of possible instructions is combinatorial in the number of different instructions (i.e., opcodes), the number of registers, the number of memory locations, and the full range of allowed symbolic values (which is exponential in the number of bits representing those values). Worse yet, the number of potential instruction sequences, that is, the size of the program space, is exponential in the length of the sequence.

Because inductive synthesis must choose a program from the space of all possible programs, this complexity is both particular and fundamental to assembly program synthesis, and not present in assembly program verification. In contrast to synthesis, verifying against a specification need only consider the single concrete assembly program involved. Correspondingly, we have observed that guessing a single program candidate from a set of input/output counterexamples is several orders of magnitude slower than verifying a concrete program against a specification.

Efficient pointer and memory handling is troublesome even for purpose-built symbolic execution engines. Rosette's generic symbolic execution engine worked well for our problem instances without memory, but with memory involved, it became slow. In a real machine, pointers are bitvector values. Reading from one in symbolic execution generates an instant explosion: a 32-bit symbolic bitvector pointer addresses 2^{32} different memory addresses and each must be considered separately. Some other more abstract representation is needed.

Hence, we now use a purpose-built symbolic execution engine and method for modeling memory. We model memory as a small set of problem-specific disjoint regions, and represent pointers as a region paired with an integer offset. At synthesis time, the integer offset is represented as a bitvector of appropriate width. This approach was inspired by memory abstractions used in C and C-related tools, such as Cyclone's [27] explicitly declared memory regions. While this change has improved our scalability in the presence of memory, we continue to face fundamental challenges as we advance our engine's heuristics.

Lesson 8: Synthesizing assembly is fundamentally hard. State is global and untyped, leading to a combinatorial explosion.

5 A Way Forward

Should we give up on OS synthesis, or is there a way forward? We believe strongly that it is too early to give up; many avenues of attack remain open.

First, much can be done at the solver level. Bitvectors and integers are fundamentally different from a logic perspective, but they are nonetheless both representations of numbers and interoperability is useful in many settings beyond just ours. We have several suggested directions for applied solver research.

One is new approaches and better support in general for mixed theories, such as for integers and bitvectors where different SMT theories can refer to the same values and where many operations and values map cleanly from one theory to the other. Another is better support for machine integers, in the sense that neither the theory of linear arithmetic on $\mathbb{Z}$ or the theory of bitvectors is the best model; rather, machine integers are $\mathbb{Z}/2^{32}$ (or 2^{64}) in which linear arithmetic is still decidable, and being able to reason about this directly would be extremely helpful. Also, Boolector [5] is a domain-specific solver for booleans and bitvectors and has been very helpful; a domain-specific solver specifically for machine integers and problems involving many machine integers would be even more helpful.

Second, at the synthesis engine level, there are several ways forward. One is to use a wider variety of target languages. Some kernel code must unavoidably be written in assembler, but traditionally much machine-dependent kernel code is written in machine-dependent C. C is semantically complicated but a restricted DSL of a similar nature would be a much less hostile environment than assembly language.

Another approach is to pursue deductive synthesis techniques [17] [18] [19] that can break large synthesis problems down into components small enough to use with CEGIS. Similarly, investigating ways to identify appropriate intermediate machine states and use those to subdivide synthesis problems would also help.

At the specification level, the critical point is that the specification problem for synthesis is the same as the specification problem for verification. Thus, we ought to be able to adopt some of the techniques that have been tried in that domain, such as layers of increasingly-detailed semi-executable specifications [9]. Specifying the behavior of machine-dependent code requires a framework for reasoning about the machine-independent abstractions that model the machine-dependent code. Then one can create a sample executable specification for an idealized machine and use it to extract predicates about the abstract state before and after.

In the case of fundamentally machine-specific operations, such as x86 segment table initialization, it will be necessary to write additional specifications to define intermediate steps in a larger machine-independent operation. This is not actually horrible. It is still better than manually writing the code that initializes the segment tables.

In the OS, it appears that the advent of synthesis changes the design tradeoffs for the interface to machine-dependent code. Traditionally, the primary consumers of the interface definition are humans writing new ports, so the interface has been designed to keep it accessible to humans: narrow (relatively few elements), high level, and perhaps excessively general with a comprehensible level of detail. A human can extract further detail when needed by referring to other existing ports and can cope with excessive generality by borrowing their code. (In fact, the traditional method for porting the BSD virtual memory system's machine-dependent module ("pmap") is to copy an existing one for a similar machine and edit it as needed.) It is less important that the interface be precise and specific than that it be easy to follow.

However, if the machine-dependent code is going to be synthesized, then the primary consumer is the code synthesis engine, and the interface design should match: it should be as low-level and specific as possible. It can be broad and freely contain many, many things; synthesizing many small things is much easier than synthesizing a few large things. This seems to indicate that the interface should be lowered, perhaps substantially, and all available excess generality squeezed out.

This leads to the final point, which is that a kernel intended to be ported by synthesis should be written for the purpose, just as existing kernels that have been verified have been written for the purpose. In particular, we intend to move forward following three principles: 1) Focus on small, formally specified OS components, 2) Target new, emerging hardware that is most in need of system software support (e.g., ASICs, FPGAs), and 3) Embrace a wide range of techniques for generating code that cover the spectrum from compilation to modern synthesis. Stay tuned for further progress.

6 Conclusion

Trying to synthesize the entire machine dependent layer of a real operating system was an audacious undertaking. We did not succeed at that, but we have, in fact, built an ecosystem of tools for synthesizing machine specific operating system functionality. Although there were many obstacles, we continue to believe it was a worthwhile undertaking and intend to take our own suggestions and undertake a slightly less ambitious project in a much more methodical fashion. We could not, however, have done so without going through the experience discussed here. We hope we have not scared potential collaborators away, but rather that others will join us in this effort.

Acknowledgments

This article is based on work supported by the U.S. Air Force and DARPA under contract FA8750-16-C-0045. The views, opinions, or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the U.S. Department of Defense or the U.S. Government.

References

[1] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=ByldLrqlx

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[3] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with Angelic Nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 339–352. https://doi.org/10.1145/1706299.1706339

[4] James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. In *Proceedings of the 2017 Programming Languages Design and Implementation Conference (PLDI '17)*.

[5] Robert Brummayer and Armin Biere. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 174–177.

[6] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[7] Yu Feng, Ruben Martins, Osbert Bastani, and Işıl Dillig. 2018. Program Synthesis Using Conflict-driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

[8] John K. Feser, Swarat Chaudhuri, and Işıl Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. *SIGPLAN Not.* 50, 6 (June 2015), 229–239. https://doi.org/10.1145/2813885.2737977

[9] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16)*. 653–669.

[10] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[11] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[12] David A. Holland, Jingmei Hu, Eric Lu, Ming Kawaguchi, Stephen Chong, and Margo I. Seltzer. 2019. *Aquarium: Cassiopea and Alewife Languages*. Technical Report. arXiv:cs.PL/1908.00093 https://arxiv.org/abs/1908.00093

[13] Lubomir Ivanov and Ramakrishna Nunna. 2001. Modeling and Verification of Cache Coherence Protocols. In *Proceedings of the 2001 IEEE International Symposium on Circuits and Systems (ISCAS)*.

[14] David Mosberger and Larry L. Peterson. 1996. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*. ACM, New York, NY, USA, 153–167. https://doi.org/10.1145/238721.238771

[15] Greg Nelson and Derek C. Oppen. 1979. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems* 1 (1979), 245–257.

[16] Amir Pnueli and Roni Rosner. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. 179–190.

[17] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 2016 Programming Languages Design and Implementation Conference (PLDI '16)*.

[18] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. In *Proceedings of the 46th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '19)*.

[19] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *OOPSLA 2015 Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126.

[20] Calton Pu, Henry Massalin, and John Ioannidis. 1988. The Synthesis Kernel. *Computing Systems* 1, 1 (1988), 11–32.

[21] Richard F. Rashid. 1986. From RIG to Accent to Mach: The Evolution of a Network Operating System. In *Proceedings of 1986 ACM Fall Joint Computer Conference (ACM '86)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1128–1137. http://dl.acm.org/citation.cfm?id=324493.325071

[22] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009. Automatic Device Driver Synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*.

[23] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-Guided Device Driver Synthesis. In *Proceedings of the 11th annual USENIX conference on Operating Systems Design and Implementation (OSDI'14)*. 661–676.

[24] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. 1984. Experience with Grapevine: The Growth of a Distributed System. *ACM Trans. Comput. Syst.* 2, 1 (Feb. 1984), 3–23. https://doi.org/10.1145/2080.2081

[25] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 136–148. https://doi.org/10.1145/1375581.1375599

[26] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[27] Nikhil Swamy, Michael W. Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe manual memory management in Cyclone. *Sci. Comput. Program.* 62, 2 (2006), 122–144. https://doi.org/10.1016/j.scico.2006.02.003

[28] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[29] Fabian Wolf, Jan Staschulat, and Rolf Ernst. 2002. Associative Caches in Formal Software Timing Analysis. In *Proceedings of the 2002 Design Automation Conference*.