InstantCheck: Checking the Determinism of Parallel Programs Using On-the-fly Incremental Hashing*

Adrian Nistor
University of Illinois at Urbana-Champaign, USA
nistor1@illinois.edu

Darko Marinov
University of Illinois at Urbana-Champaign, USA
marinov@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign, USA
torrella@illinois.edu

ABSTRACT

Developing multithreaded programs in shared-memory systems is difficult. One key reason is the nondeterminism of thread interaction, which may result in one code input producing different outputs in different runs. Unfortunately, enforcing determinism by construction typically comes at a performance, hardware, or programmability cost. An alternative is to check during testing whether code is deterministic.

This paper presents *InstantCheck*, a novel technique that checks determinism with a very small runtime overhead while requiring only a minor hardware extension. During code testing, InstantCheck can check whether the code under test ends up in a deterministic state in various runs. The idea is to compute a 64-bit hash of the memory state and compare the hashes of different test runs that have the same input. If two runs have different hashes, InstantCheck reports state nondeterminism. For efficient operation, InstantCheck uses *on-the-fly incremental hashing in hardware*. The hash is kept in a per-core 64-bit register, which trivially supports virtualization, migration, and context switching.

We use InstantCheck to understand the determinism properties of 17 popular applications, including Sphinx3, PBZip2, PARSEC, and SPLASH-2. InstantCheck incurs a negligible average runtime overhead of 0.3% over native testing runs. We also show how using InstantCheck programmers can find bugs and discuss other applications of fast memory-state hashing. While using InstantCheck, we found a *real bug* in the widely used PARSEC benchmark.

1. INTRODUCTION

Parallel programming in shared-memory systems is a challenging task. Often, code is written with threads that race over synchronization or data variables, changing the memory state with non-commutative updates. In some cases, a piece of code can produce, for the same input memory state, different resulting memory states in different runs with different thread interleavings. Such code, which we refer to as *Externally* nondeterministic (or nondeterministic for short), is hard to debug, test, maintain, and reuse.

Determinism is increasingly recognized as an important property for a large body of parallel code [3,6,9,10,12,41,44]. Most of the recent work [3, 6, 12, 41, 44] achieves external determinism by requiring a more restrictive property, namely fixing or strictly limiting the order(s) of inter-thread communications — an environment we refer to as having *Internal* determinism. Unfortunately, enforcing internal determinism usually comes at a performance, hardware, or programmability cost. For example, it may be attained by using a non-standard hardware architecture [12] or runtime system [3, 41] that forces a fixed communication order, by limiting the types of thread synchronization primitives allowed in the code [44], or by annotating the code and limiting the constructs allowed [6], possibly hindering general-purpose programming. Burnim and Sen [9, 10] propose checking of determinism which focuses on selected data structures in a program and by default checks neither internal nor external determinism.

Efficient checking of external determinism: Instead of *enforcing internal determinism*, we propose *InstantCheck*, a novel technique for *checking external determinism* during testing of parallel code. For such testing, developers typically run the code many times for the same input, using random, stress, or systematic testing to cover different thread interleavings [8,35,36,42,45,48]. During these testing runs, the code is being checked for several correctness properties such as crashes, assertion violations, semantic bugs, deadlocks, data races, etc. InstantCheck piggybacks on the testing already done for parallel code and simply adds several very fast checks for external determinism.

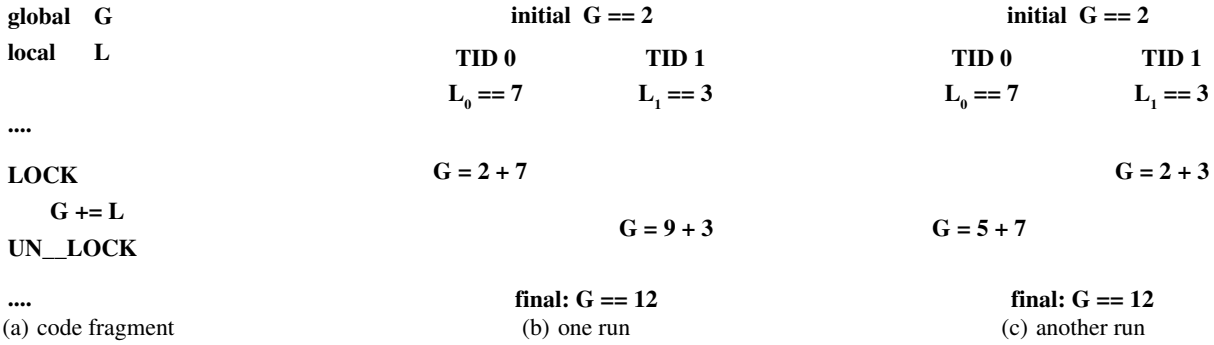
InstantCheck checks whether different runs with the same input state produce deterministic output state. To efficiently compare states, InstantCheck *distills the memory state into a 64-bit hash* and compares the hashes for different test runs. If hashes differ for two runs, InstantCheck reports nondeterminism. InstantCheck uses hashes because they are efficient to compute and yet extremely accurate: false positives (i.e., two memory states are the same but their hashes differ) are not possible, and false negatives (i.e., two memory states differ but their hashes are the same) are statistically rare — for a 64-bit hash, the probability is 1 in 2^{64}.

We propose two schemes for computing hashes. (1) Our main contribution is *InstantCheck*$_{Inc}$[1], a scheme that uses *incremental hashing* [2] to compute hashes *on-the-fly* as the code writes to memory. We present two versions of this scheme: a hardware-supported version, *HW-InstantCheck*$_{Inc}$, which has a very low runtime overhead, and a software-only version, *SW-InstantCheck*$_{Inc}$, which requires no new hardware. (2) We also present another, software-only scheme, *SW-InstantCheck*$_{Tr}$, that does not compute hashes on-the-fly but rather traverses the entire memory state.

InstantCheck does not suffer the performance or programmability penalties associated with enforcing internal determinism. In fact, InstantCheck does not require the code to be internally deterministic: in different runs, the code may have different inter-thread communication, different intermediate states, or even (benign) data races, as long as the runs end up with the same state.

*This work was supported in part by the National Science Foundation under grants CCF 09-16893, CCF 07-46856, and CNS 07-20593; and Intel and Microsoft under the Universal Parallel Computing Research Center (UPCRC).

[1]The name "InstantCheck" signifies that the incremental scheme makes the memory state hash instantly available for comparison and determinism checking whenever needed.



(a) code fragment (b) one run (c) another run

Figure 1: Example of external determinism. The final state is deterministic, even though the intermediate steps are not. (a) Code with a global variable G updated using a local variable L of each thread. (b) Run of code with thread 0 first updating G. (c) Run of code with thread 1 first updating G.

Minor hardware requirement: HW-InstantCheck$_{Inc}$ implements on-the-fly incremental hashing in hardware, using a per-core 64-bit register and a hashing module. This minimal hardware trivially supports virtualization, migration, and context switching, and scales with the number of cores. Incremental hashing enables a high-performance hardware implementation because: (1) the operations on the hashing register are core-local and (2) the local operations on each hashing register can be performed in parallel and out-of-order. The hardware extension proposed for HW-InstantCheck$_{Inc}$ can be also used for several other software development tasks such as detecting software bugs, separating benign data races, improving the efficiency of testing, and reducing the cost of deterministic replay.

Evaluation: We use InstantCheck to understand the determinism properties of 17 applications: `sphinx3` [26], `pbzip2` [17], and applications from PARSEC [4] and SPLASH-2 [50]. These applications are challenging for determinism checking because they were *not written for determinism* but for high-performance, and they use a variety of thread communication constructs, global variables, benign races, and floating point (FP) operations. InstantCheck shows that 14 of the 17 applications are externally deterministic when allowing for FP imprecision and some small nondeterministic data structures. In contrast, most of this code is *not* internally deterministic and thus cannot be checked by the techniques based on internal determinism [6, 44] and does not need to have internal determinism enforced [3, 12, 41]. HW-InstantCheck$_{Inc}$ incurs a negligible average runtime overhead of 0.3% over native testing runs. While the goal of our experiments was to characterize determinism and not look for new bugs, we still found a concurrency bug in the widely used PARSEC benchmark [4]. The PARSEC author corrected the bug after we reported it.

2. INSTANTCHECK

2.1 The Idea and Example

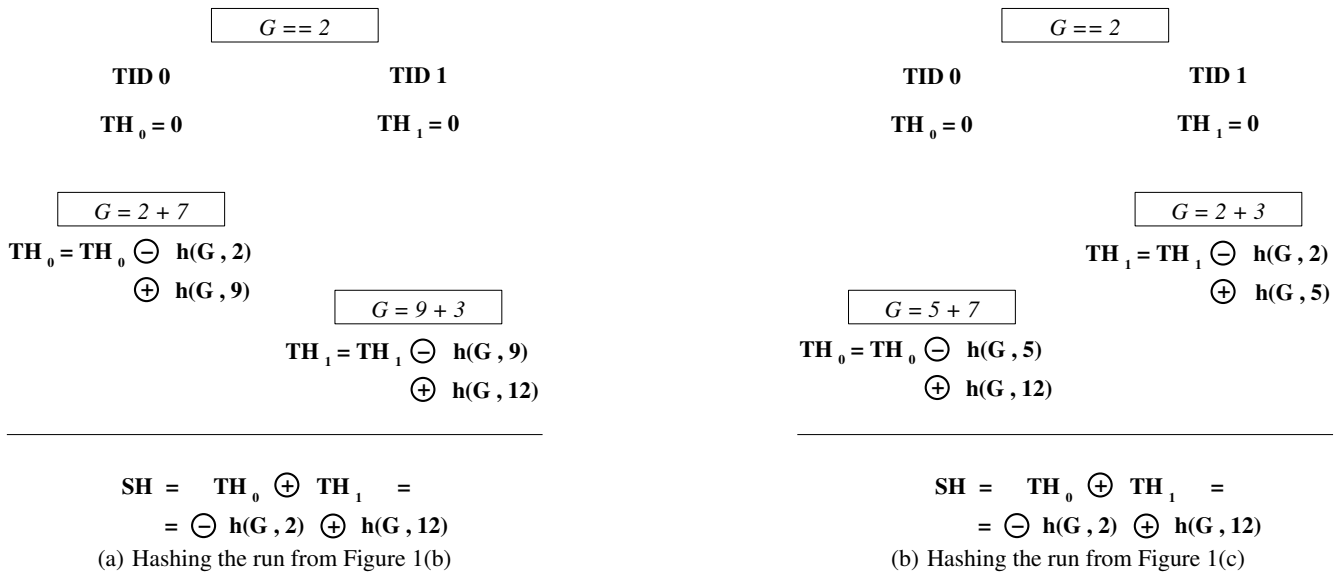
InstantCheck is based on the following observation: if execution leaves a program in a *deterministic state*, then the program is *externally* deterministic, regardless of how the program executed *internally*. During testing, as parallel code is run many times for one input [8,35,36,42,45,48], InstantCheck adds several very fast checks to detect if different runs lead to deterministic states. These additional checks are done simultaneously with the checks for regular correctness properties and do not require additional test runs. InstantCheck summarizes the memory state (heap and static data) in a 64-bit hash, which HW-InstantCheck$_{Inc}$ computes very quickly using a minor hardware extension (Section 3), while SW-InstantCheck$_{Inc}$ and SW-InstantCheck$_{Tr}$ compute without hardware support but slower (Section 4). For all schemes, InstantCheck uses software to compare the computed hashes and to control sources of input nondeterminism (Section 5). If two runs produce different hashes, InstantCheck found some nondeterminism in the code. If many runs produce the same hash, the code is deterministic within the coverage of the test.

Figure 1(a) shows an example that illustrates the difference between internal and external determinism. This code fragment is a simplified version of code appearing in several PARSEC and SPLASH-2 applications. The shared, global variable G is updated using the local variable L of each thread. Figure 1(b) shows the run where G is updated first by thread 0 and then by thread 1, while Figure 1(c) shows the run where G is updated first by thread 1 and then by thread 0. The final states are the same, with G being 12. Thus, any other code using G after the run ends sees the code in Figure 1(a) as deterministic. However, note that the code executes internally nondeterministically: the two runs in figures 1(b) and 1(c) update G in different orders, the intermediate values of G differ (9 or 5), and the final value (12), although the same, is written by different threads (0 or 1). InstantCheck successfully ignores this internal nondeterminism and correctly reports the code as being externally deterministic (within the test coverage). Note that if G and L are floating point variables, different order of additions could result in different results; Section 5 discusses how InstantCheck handles this and other sources of bit-by-bit nondeterminism.

2.2 Capturing State using Incremental Hashing

The key to HW-InstantCheck$_{Inc}$'s efficiency is to compute the memory state hashes *incrementally*, updating the hash value each time a memory location is modified. Thus, HW-InstantCheck$_{Inc}$ avoids the cost of traversing the shared memory (heap and the static data) each time it needs to compute the hash. Bellare and Micciancio [2] introduced incrementally computable hash and formally proved it has the same good properties in terms of low hash collision as non-incrementally computable hashes. We describe here how HW-InstantCheck$_{Inc}$ uses incremental hashing to capture the memory state, and how HW-InstantCheck$_{Inc}$ computes such a hash in a multicore system using core-local operations.

If a program memory state S has values v_1, v_2, ..., v_m at addresses a_1, a_2, ..., a_m, then we define its *State Hash* as $SH(S) = h(a_1, v_1) \oplus h(a_2, v_2) \oplus \ldots \oplus h(a_m, v_m)$, where $h(a, v)$ is the hash



(a) Hashing the run from Figure 1(b)

(b) Hashing the run from Figure 1(c)

Figure 2: Capturing program state with incremental hashing. (a) Hashing operations corresponding to the run in Figure 1(b). (b) Hashing operations corresponding to the run in Figure 1(c).

of the address and value of one memory location (e.g., computed by CRC), and $\oplus$ is a 64-bit modulo addition. The hash includes addresses such that a different permutation of the same values does not have the same State Hash. State Hash can be computed incrementally, as the program executes, rather than traversing all memory locations whenever the memory hash is needed. This is because modulo addition is commutative and associative, and modulo subtraction can be used to cancel the effect of modulo addition. For example, suppose that during execution, one memory state S with a value v at an address a is updated to S' by writing a new value v' to the address a. Then the hash of S' can be generated simply as $SH(S') = SH(S) \ominus h(a, v) \oplus h(a, v')$, where $\ominus$ is modulo subtraction.

Figures 2(a) and 2(b) show how HW-InstantCheck$_{Inc}$ hashes the memory state of the runs in figures 1(b) and 1(c), respectively. In the figure, TH_0 and TH_1 are *Thread Hashes* for threads 0 and 1, respectively. SH is the State Hash for the memory state, and h is a regular hash function (e.g., CRC) used to hash individual memory locations. For example, $h(G, 2)$ hashes the address of variable G and its value 2.

When the execution starts, HW-InstantCheck$_{Inc}$ sets SH, TH_0, and TH_1 to zero. For each write to a memory location, HW-InstantCheck$_{Inc}$ subtracts/adds h of the old/new value from/to the Thread Hash. For example, consider Figure 2(a) when thread 0 writes value 9 to variable G, which was previously 2. HW-InstantCheck$_{Inc}$ subtracts $h(G, 2)$ from TH_0 and adds $h(G, 9)$. These are core-local operations; each core can see the old value at a given memory location because in a typical cache-coherent machine, a write access first brings the cache line with the current values into the processor's cache and only then updates the cache line. When the State Hash needs to be computed, HW-InstantCheck$_{Inc}$ computes in software SH as the modulo sum of TH_0 and TH_1. In this example, $SH = TH_0 \oplus TH_1 = \ominus h(G, 2) \oplus h(G, 12)$ for both different executions in figures 2(a) and 2(b). SH reflects the final state ($G == 12$) because the input is fixed ($G == 2$). It is interesting to observe that the per-thread hashes TH_{thID} can have *different values* for different runs even when SH has the *same value*; effectively, this case corresponds to internal nondeterminism that results in external determinism.

Incremental hashing has a highly efficient hardware implementation. Each processor computes *locally* the Thread Hash TH_{thID}, and only when the State Hash is needed, all the TH_{thID} are combined into SH (Figure 2). The TH_{thID} Thread Hashes are computed in hardware because they are updated frequently. SH is computed in software; although it is a global operation, it is extremely rare (e.g., 10 – 10000 times in a program run) relative to the total program execution. Furthermore, State Hash is typically computed at barriers, and thus summing up the Thread Hashes overlaps with waiting at the barrier.

Note that state comparison and hashing need not compare the *entire* memory state, but rather the programmer can select to *ignore some* memory locations. Incremental hashing allows HW-InstantCheck$_{Inc}$ to easily remove some memory locations from the hash computation. To do so, HW-InstantCheck$_{Inc}$ simply adds the hashed initial value of each ignored memory location and subtracts the hashed current value. For example, in Figure 2, HW-InstantCheck$_{Inc}$ can ignore variable G by simply doing: $SH = SH \oplus h(G, 2) \ominus h(G, 12)$. This deletes the value of G from SH. Deleting (potentially nondeterministic) variables allows automatic checking of determinism for the rest of the program state.

2.3 Bug Detection with InstantCheck

InstantCheck can detect bugs whenever a code that the programmer expects to be deterministic causes nondeterministic behavior. This covers a large number of very different bug types, including semantic bugs, atomicity violations [13, 28, 52], order violations [27, 29], and data races [31, 38, 45]. InstantCheck detects all these different types of bugs because it detects the nondeterministic *effects* of the bug and not particular characteristics that vary from one bug type to another. In contrast, existing testing techniques typically target only one of these bug types.

Detecting bugs with InstantCheck works as follows. InstantCheck checks determinism at each program barrier and at run end.

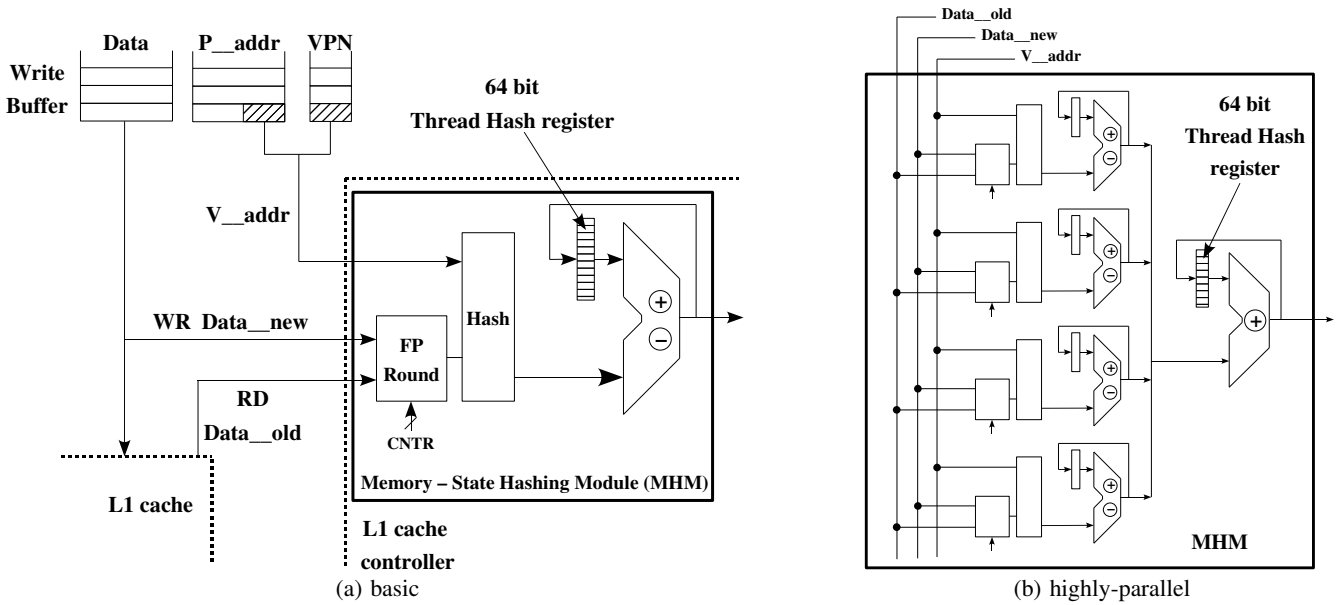


Figure 3: High-level design of the Memory-State Hashing Module (MHM): (a) basic and (b) highly-parallel designs.

This is done automatically by intercepting *pthread* barrier calls (Section 5). Barriers are natural and intuitive points for a deterministic program to be in a deterministic state as shown by previous work [9, 12, 44] and our experiments (Section 7), since barriers usually delineate program phases. The programmer may also specify additional program points where she expects her program to be in a deterministic state (e.g., the end of some loop iterations or hand-coded barriers). With HW-InstantCheck$_{Inc}$, she can afford to check determinism at as many points as desired with no fear of slowing down multiple testing runs.

InstantCheck only finds the presence of a bug but does not localize where in the code the bug is. Based on our experience with InstantCheck (both debugging the InstantCheck implementation itself and using it to understand nondeterminism in applications), we found the following tool-supported methodology to be very helpful in bug localization. When InstantCheck detects a nondeterministic program point, the programmer knows the region of code between the last deterministic point and the first nondeterministic point. If the programmer expects this region to be deterministic (according to the algorithm), then the programmer needs to debug the execution of that region to localize the bug.

We developed a simple prototype tool to help in this debugging. The tool re-executes the program for the two runs that differ. During these two re-executions, the tool stores the entire memory states (rather than just state hashes) at the point where InstantCheck found nondeterminism. The tool then compares the two states created by the two runs and detects the memory addresses that differ. The tool maps each of these addresses back to the source code line that allocated the address and to the offset of the address from the start of the allocation block (e.g., array index or struct field). (In our current prototype, we manually instrument the allocation sites in the source code to dynamically track (1) which memory address is allocated at what code line and (2) the type of the allocated memory, but this instrumentation can be automated.) The tool finally reports the allocation sites and offsets to the programmer. The programmer now knows the part of memory that behaved nondeterministically, in addition to the region of code where the nondeterminism manifested, and can use a favorite debugging technique (e.g., breakpoints, watchpoints, *printf*, etc.) to see why those parts of memory behaved nondeterministically in that region of code.

3. HARDWARE SYSTEM

3.1 Hardware Support

For incremental hashing, HW-InstantCheck$_{Inc}$ uses a *Memory-State Hashing Module* (MHM) that is part of the L1 cache controller of each core. Figure 3(a) shows a high-level design of the MHM. All MHM operations are local to each core, with no inter-core communication or synchronization.

The main part of the design includes a hash unit (computing the function h discussed in Section 2.2), a modulo add/subtract unit, and a 64-bit *Thread Hash* (TH) register that accumulates the hashed writes performed by the thread. When the write buffer updates the L1 cache with a new value (*Data__new*) for a variable at virtual address *V__addr*, the MHM reads the old value (*Data__old*) in the cache and computes $TH = TH \ominus hash(V\_addr\ , Data\_old) \oplus hash(V\_addr, Data\_new)$. Therefore, we feed V_addr, $Data\_old$, and $Data\_new$ to the MHM. Obtaining $Data\_old$ does not incur an additional cache miss in write-allocate caches (ubiquitous in current general purpose processors), because either the data is already in the cache or will be brought any way to service the write. To obtain V_addr, we proceed as follows. When a write instruction retires from the ROB, as the data and its physical address (P_addr) are saved in the write buffer structure, the hardware also saves the virtual page number (*VPN*) of the address (Figure 3(a)). With *VPN* and the page offset from P_addr, the hardware can later compute V_addr when the write is pushed into the L1 cache.

As described in Section 5, we may want to round-off the floating point (FP) values before hashing. The goal is to eliminate the small differences in FP values that appear in different runs due to FP precision limitations. Consequently, the MHM has an FP round-off unit in front of the hash unit, and the $Data\_old$ and $Data\_new$ wires are routed through it. If the write-buffer entry

currently pushed into the L1 was generated by an FP store instruction (Section 5), the *CNTR* input in the FP round-off unit enables the rounding-off hardware.

In a design for expert numerical programmers, other *CNTR* inputs can select what type of rounding operation is performed, together with some programmer-settable parameters for the rounding operation. Specifically, we propose to give programmers two alternatives: zero-out the least-significant *M* bits of the mantissa, or take the floor to the number with only *N* decimal digits. The first alternative is used when the programmer wants to discard small *relative* differences between program outputs; the second when she wants to discard small *absolute* differences. Implementation-wise, the first alternative is simpler, since it involves logically AND-ing the mantissa with a mask. The second alternative is similar to an x86 rounding instruction and is used in systematic testing [48].

The HW-InstantCheck$_{Inc}$ hardware trivially supports virtualization, migration, and context switching (just save and restore a register), and it is highly scalable. Each core has its own MHM module, which computes *locally* the hash in the TH register without any inter-core communication. When the overall *State Hash* is desired, software computes it by simply modulo adding the TH registers (Section 2.2); this is a global but very infrequent operation (about 10 – 10000 times in a program run). Typically, InstantCheck checks for determinism at barriers and thus the gathering of TH values overlaps with the barrier communication.

3.2 Flexible Implementation Choices

The MHM can be implemented in different ways depending on whether the goal is to optimize for speed or area. The key insight is that incremental hashing uses modulo addition, which is both *commutative* and *associative*, for both the same and different addresses. This means that the hashing operations accumulated into the TH register can occur in any order. Moreover, they can be performed in parallel in different clusters, where partial results are accumulated in local cluster registers and only later on merged into the TH register. There are no limits to how many operations can occur in parallel, so each individual operation can be as slow or as fast as required by speed and area constraints.

As an example, while Figure 3(a) shows a design optimized for area, Figure 3(b) shows one optimized for performance through parallelism. In this design, the MHM has several clusters, and each pair (*Data*, *V_addr*), where $Data$ is $Data\_old$ or $Data\_new$, is sent to any one cluster. The pairs containing *Data_old* and *Data_new* can even go to different clusters. Each cluster computes a partial sum of hashes. These results are later summed up in the TH register.

Moreover, the writes in the write buffer can be drained and sent to the MHM in any order. Similarly, *Data_old* does not need to be sent to the MHM right before *Data_new*, as long as *Data_old* was captured correctly. For example, *Data_old* could be sent much earlier than *Data_new*, or even after it. Such flexibility enables the use of the design that is easiest to implement.

3.3 MHM Software Interface

MHM interfaces to the software through the assembly instructions of Figure 4. The *start_hashing* and *stop_hashing* instructions enable and disable the MHM hashing operation. They allow for code, such as an analysis tool, to be run in the checked thread's address space without interfering with the determinism checking.

The *save_hash* and *restore_hash* instructions save and restore the TH register to and from memory. This enables virtualization, and threads can be context-switched or migrated between cores. Indeed, multiple programs may be tested for determinism simultaneously on the same multiprocessor. The OS or VM monitor simply saves and restores a thread's TH register at thread switching points, just like it does for any other register. There is no additional overhead for virtualization, context switching, or migration.

Instruction	Description
start/stop_hashing	Start/stop hashing the values of the memory writes
save/restore_hash *addr*	Save/restore the TH register to/from memory location *addr*
minus_hash *addr*	Subtract the hash of the current value of the memory at *addr* from TH
plus_hash *addr val*	Add to TH the hash of *val* as if *val* would be the current value at memory location *addr*
start/stop_FP_rounding	Start/stop rounding-off FP values before hashing

Figure 4: MHM software interface.

The *minus_hash* instruction subtracts from the TH register the hash of the current value at a memory address. The *plus_hash* instruction adds a given value to the TH register. Combined, these operations eliminate the effect of an address from the hash value. As described in Section 2.2, this enables the programmer to explicitly exclude the effect of some nondeterministic variables.

The *start_FP_rounding* and *stop_FP_rounding* instructions specify whether FP values should be rounded-off before hashing. By rounding, the hashes are unaffected by the small differences in FP values that appear in different runs due to FP precision limitations. We allow the programmer to specify this behavior because different programmers have different requirements. Moreover, expert numerical programmers may be given the choice of selecting what type of rounding-off operation to perform — zero-out *M* mantissa bits or take the floor to the number with only *N* decimal digits — and even providing the values for the *M* and *N* parameters.

4. HASHING IN SOFTWARE

HW-InstantCheck$_{Inc}$ performs in hardware the most expensive operation for incremental hashing, namely computation of Thread Hash values. We next present SW-InstantCheck$_{Inc}$, which computes Thread Hash values in software. We then present SW-InstantCheck$_{Tr}$, which computes state hashes in software by traversing the memory state. We finally present hashing for I/O.

4.1 Incremental Hashing in Software

SW-InstantCheck$_{Inc}$ is a software-only version of HW-InstantCheck$_{Inc}$. SW-InstantCheck$_{Inc}$ instruments the code under test to add hashing operations for each store instruction. These operations read the old value for the destination location to subtract its hash and add the hash of the new value being stored, as illustrated in Figure 2. The instrumentation is straightforward with one caveat about atomicity. If the instrumented operations are executed atomically with the store (to ensure that the proper old value is read), the execution overhead could be high. If the instrumented operations are not executed atomically with the store, then in the presence of write-write races in the code under test, a stale old value could be read. This stale value could affect the hash computation and report nondeterminism even if the code is deterministic. The programmer has to decide whether to pay a higher overhead penalty or to accept potential false alarms; HW-InstantCheck$_{Inc}$ is not only faster than SW-InstantCheck$_{Inc}$ but also does not burden the programmer with this decision because HW-InstantCheck$_{Inc}$ atomi-

cally accesses old and new values in the L1 cache (Section 3.1). Our implementation of SW-InstantCheck$_{Inc}$ used for experimental evaluation (Section 7) serializes program execution and achieves atomicity without using locks.

4.2 Non-Incremental Hashing in Software

The SW-InstantCheck$_{Tr}$ scheme computes the state hash by *traversing* the entire static data and heap. To traverse the state, SW-InstantCheck$_{Tr}$ needs to know which addresses were dynamically allocated (*malloc*ed but not *free*d) and which addresses store *float* and *double* values (to perform rounding as described in Section 5). Note that SW-InstantCheck$_{Inc}$ updates the hash for each store instruction, and, therefore, it does not need to traverse the state and can decide what values are FP based on the instruction and not necessarily based on the address.

To track the FP values inside dynamically allocated memory in C/C++, SW-InstantCheck$_{Tr}$ annotates each allocation site with type information. This type information encodes for each allocated byte whether it is a start of *float*, *double*, or some other type, and for structs and arrays it encodes recursively the types of their elements and the length. In our current SW-InstantCheck$_{Tr}$ prototype, we add these annotations manually. Two challenges complicate automation of this annotation task: (1) application-specific functions that wrap *malloc/free*, which is a coding practice used to enhance error checking and modularity [16]; and (2) starting the usage of allocated memory several bytes after the first allocated byte to improve cache behavior. HW-InstantCheck$_{Inc}$ is not only faster than SW-InstantCheck$_{Tr}$ but also more automated as it does not require these annotations.

During execution, SW-InstantCheck$_{Tr}$ maintains a *table of allocated blocks with their type information*. SW-InstantCheck$_{Tr}$ adds a new entry to the table for each *malloc* and removes an entry for each *free* (or application-specific wrappers). Compared to the native execution, SW-InstantCheck$_{Tr}$ has the overhead for maintaining this table of allocated blocks and the overhead for traversing the state, which requires lookups into the table of allocated blocks and also incurs cache misses as it sweeps through the entire memory. We do not use our SW-InstantCheck$_{Tr}$ prototype for performance comparison with HW-InstantCheck$_{Inc}$ (Section 7.3) because we did not optimize our SW-InstantCheck$_{Tr}$ prototype; instead, we compare HW-InstantCheck$_{Inc}$ against an ideal, traversal-based hashing that ignores most of the overhead of SW-InstantCheck$_{Tr}$. The results show that HW-InstantCheck$_{Inc}$ is still much faster than such an ideal, traversal-based hashing. We do use our SW-InstantCheck$_{Tr}$ prototype to confirm the determinism results from our HW-InstantCheck$_{Inc}$ implementation.

4.3 I/O

InstantCheck focuses on memory state determinism, but for completeness we discuss I/O determinism. We implement in software a limited version for checking the determinism of outputs. The obvious approach is to compute a hash on the total output stream. A full implementation would do this at a point in the *libc* library or the OS kernel where the partial outputs from various threads can no longer be reordered in buffers. In our current implementation, we changed the *write* function from *libc* to hash the actually written bytes before the return from the function, which fully captures only the behavior of properly-synchronized outputs.

5. CONTROLLING SOURCES OF NONDETERMINISM

Various InstantCheck versions compute state hashes in hardware or software, but InstantCheck always compares these hashes in software and also controls sources of nondeterminism in software. By default, InstantCheck computes the hash of the entire state, and if any bit of two states differs, the hashes differ, and InstantCheck reports nondeterminism. To ensure that any nondeterminism can only come from different thread interleavings, InstantCheck automatically controls other nondeterminism sources such as dynamic memory allocation, nondeterministic library calls, and FP roundoff.

Dynamic memory allocation can produce nondeterministic states since calls to *malloc* can return different addresses in different runs. However, most of the time, the user would want to ignore this nondeterminism. InstantCheck automatically handles this, effectively treating addresses returned by *malloc* as program input and capturing it as done for deterministic replay [33, 48]. More precisely, InstantCheck logs the addresses returned by the dynamic allocator in the previous runs and repeatedly returns the same addresses for future runs. InstantCheck automatically intercepts the calls to the dynamic memory allocator and does not require the programmer to change the code. When InstantCheck intercepts an allocation call, it also sets the values inside the allocated region to zero (as *calloc* does). This ensures that the initial state of that memory is fixed: all zeroes. In contrast, if the memory contained uninitialized values, it could have corrupted the hash and resulted in false positives for reported nondeterminism.

Nondeterministic library calls such as *gettimeofday* or *rand* return different values each time they are called. Thus, on multiple runs, they will return different results. InstantCheck, as most systems [21, 33, 48], treats the results of these calls as input, and ensures that the calls return the same values for successive runs. As any other input, the results of these calls can be varied in tests, to increase coverage.

Floating point (FP) operations can lead to different results in parallel code when non-associative FP operations are executed in different orders in different runs. For example, the runs from figures 1(b) and 1(c) can produce different values for G if G and L are FP variables. Since some users may consider such changes nondeterministic, while others may want to ignore them, InstantCheck allows the user to specify whether to compare FP numbers bit-by-bit or to ignore some differences. If the user chooses to ignore differences, InstantCheck rounds the FP values as described in Section 3.1. By default, InstantCheck rounds to the closest 0.001, as typically done in systematic testing [49]. The MHM module needs to know which writes are to FP values. In the current implementation, InstantCheck uses the LLVM compiler [23] to mark the FP writes. Since this process is done by the compiler, it is fully automated for the user. In theory, this approach could miss some FP values, e.g., in some unions. However, although our experiments use code with unions, we never encountered such a case.

Auxiliary structures that are used during computation but are not considered result may be left in nondeterministic states. For example, a list of free "task nodes" in SPLASH-2's `cholesky` may have the nodes linked in any order and containing some old values. From the programmer's functional view, the nodes are free and their values do not matter. However, the memory state is different, and InstantCheck correctly detects this as nondeterminism. Silently ignoring such nondeterminism could be dangerous, because the programmers may mistakenly rely on the supposedly deterministic behavior of the code. For advanced users, InstantCheck allows *explicitly* specifying nondeterministic structures. InstantCheck then automatically deletes such structures from the hash using the technique described in Section 2.2.

Software bugs can cause a supposedly deterministic program to have nondeterministic behavior. This enables InstantCheck to detect different types of bugs, as described in Section 2.3. In fact,

during our experiments, we found a real bug in the widely used PARSEC benchmark. We found this bug while investigating the nondeterminism that InstantCheck reported in a part of code we expected to be deterministic.

Truly nondeterministic algorithms such as heuristic search or optimization algorithms (e.g., simulated annealing) can give different results for different runs. InstantCheck correctly detects such code as nondeterministic, for example a simulated annealing implementation from PARSEC [4]. An interesting case we analyze in Section 7 is a Monte Carlo simulation from PARSEC which one would expect to be nondeterministic. However, this particular code is deterministic, because each thread has a *local* random number generator; the nondeterminism is manifested only when the random seeds change and is not due to parallel execution.

6. OTHER APPLICATIONS OF THE HARDWARE PRIMITIVE

Fast comparison of memory states is a powerful primitive. In addition to checking for determinism, it can benefit several other applications such as filtering out benign data races, systematic testing, and deterministic replay.

6.1 Filtering Out Benign Data Races

Data races are common and difficult to expose concurrency bugs. Their detection has been researched for years with several recent techniques proposed from both the software and hardware community [14, 18, 31, 37, 40]. Most techniques report even benign races, which result in deterministic output. Narayanasamy *et al.* [38] report that 90% of races are benign and show how to filter out benign races by comparing the memory states produced when flipping the race. Their approach could benefit from the use of InstantCheck, which provides a fast state comparison. Note that using InstantCheck to detect races, as described in Section 2.3, already filters out benign races because of the state comparison that InstantCheck performs.

6.2 Systematic Testing

Systematic testing [36, 48] is a testing technique that *systematically* executes different thread interleavings of a given program, while checking for bugs. For example, CHESS [36] systematically explores all interleavings with a small number of preemptive thread context switches. CHESS is a popular testing tool used on a daily basis at Microsoft. In a comparison to stress testing [36], CHESS found and was able to reproduce 25 bugs that stress testing did not find or could not reproduce for many months. Unfortunately, the search space of systematic testing grows exponentially with the number of possible context switches. One way to combat that search-space explosion is to identify equivalent states and prune the search. Comparing entire states in software is expensive, and so CHESS does not perform it. Instead, it only compares the happens-before relationship [22] which is an approximation that can miss equivalent states. For example, the two runs in Figure 1 lead to the same state but have different happens-before. Using InstantCheck to check state equality (instead of happens-before) can speed up systematic testing of C/C++ code (as it enables better state pruning) and make it more precise (as it detects different states even when the synchronization order is the same).

6.3 Deterministic Replay

Deterministic replay [19, 33, 53] faithfully re-executes a program run, e.g., for debugging. The classic approach is to save a detailed execution log for later replay. Recent techniques [1, 24, 25, 43] achieve low runtime overhead by saving only a *partial* log. During replay, they search many executions that obey the log to see which one recreates the bug manifested in the original execution. Using InstantCheck to check state equality can assist these techniques to detect when they reproduce the entire state, not only the bug. This enables the programmer to inspect all the variables of the program as they were in the original run. Additionally, the state hash can be a part of the partial log saved by the system, which allows early detection of a replay that does not obey the log. InstantCheck makes this feasible as it adds small runtime overhead to both the original execution and replay.

7. EVALUATION

7.1 Experimental Setup

We model HW-InstantCheck$_{Inc}$'s lightweight hardware support using Pin [30]. We also implement SW-InstantCheck$_{Inc}$'s instrumentation using Pin. We implement SW-InstantCheck$_{Tr}$'s maintenance of the table of allocated blocks (with their type information) and the state traversal in C++.

We evaluate InstantCheck using a testing technique which serializes thread execution, i.e., a thread scheduler runs one thread at a time and switches between threads at synchronizations. This approach is used by the latest tools from Microsoft Research (PCT [8], CHESS [36]) because it exposes the parallel execution complexity much better and much faster than stress testing with a truly parallel execution [8, 36]. In our evaluation, the thread to run is chosen randomly. This random thread scheduler is not a part of InstantCheck: in real usage, this scheduler would be replaced by the tool that the programmer already uses for testing. For example, if the programmer uses stress testing, there would be no scheduler at all, while if the programmer uses PCT or CHESS, the scheduler would be replaced by PCT or CHESS.

We use InstantCheck to understand the determinism properties of 17 applications: the `sphinx3` [26] speech recognition software, the `pbzip2` [17] open source file compressor, and applications from PARSEC [4] and SPLASH-2 [50] suites. These applications are challenging for determinism checking because (1) they were *not written for determinism* but for high performance, and (2) they have a variety of thread communication constructs, global variables, benign races, and FP operations. All tests use 8 threads with the following inputs: `sphinx3` uses the *pittsburgh* utterance, `pbzip2` compresses one of our log files, the PARSEC applications use *simmedium* input, and the SPLASH-2 applications use the default input. We use the newer `pbzip2` version 1.0.5 that does not contain the bug reported in [55]. For *evaluation purposes, we test 30 runs* per application and interpret all results within the test coverage provided by these 30 runs. However, as shown in Section 7.2, *nondeterminism is often detected after just 2 or 3 runs*, and in real usage of InstantCheck, the programmer can stop as soon as nondeterminism is detected.

7.2 Determinism Checking with InstantCheck

7.2.1 Characterizing Determinism

Table 1 shows the determinism characteristics detected with InstantCheck. We group the applications into four types of determinism as shown in the table which separates groups by horizontal lines. Column 4 shows if the application has floating-point operations or not. Columns 5, 7, 9, and 12 show if the application is bit-by-bit deterministic, deterministic with FP rounding, deterministic if ignoring some small structures, or nondeterministic. Column 6 shows the first test run when the applications are detected as bit-by-

Det Type	Application	Source	FP?	Det as is ?	First NDet. Run	Impact of FP rounding on Det.	First NDet Run after FP	Isolating Small Structs on Det.	# Dyn Checking Points		Det. at the End
									Det	NDet	
bit by bit	blackscholes	parsec	Y	Y	–	Det → Det	–	–	101	0	Y
	fft	splash2	Y	Y	–	Det → Det	–	–	13	0	Y
	lu	splash2	Y	Y	–	Det → Det	–	–	68	0	Y
	radix	splash2	N	Y	–	Det → Det	–	–	12	0	Y
	streamcluster	parsec	Y	Y	–	Det → Det	–	–	12928	**74** ⋆	Y
	swaptions	parsec	Y	Y	–	Det → Det	–	–	2501	0	Y
	volrend	splash2	N	Y	–	Det → Det	–	–	6	0	Y
FP prec	fluidanimate	parsec	Y	N	2	NDet → Det	–	–	41	0	Y
	ocean	splash2	Y	N	3	NDet → Det	–	–	871	0	Y
	waterNS	splash2	Y	N	3	NDet → Det	–	–	21	0	Y
	waterSP	splash2	Y	N	2	NDet → Det	–	–	21	0	Y
small struct	cholesky	splash2	Y	N	3	NDet → NDet	3	NDet → Det	4	0	Y
	pbzip2	openSrc	N	N	2	NDet → NDet	2	NDet → Det	1	0	Y
	sphinx3	alpBench	Y	N	2	NDet → NDet	2	NDet → Det	4265	0	Y
NDet	barnes	splash2	Y	N	2	NDet → NDet	2	–	2	16	N
	canneal	parsec	N	N	2	NDet → NDet	2	–	0	64	N
	radiosity	splash2	N	N	2	NDet → NDet	2	–	0	19	N

Table 1: Determinism characteristics. As with any dynamic testing tool, the results are valid within the test coverage. ⋆ denotes the 74 nondeterministic barriers in `streamcluster` caused by the bug that we detected using InstantCheck; when the bug is fixed, these 74 barriers also become deterministic.

bit nondeterministic, and column 8 shows the first test run when the applications are nondeterministic even after applying FP rounding. These numbers are a measure of how fast the programmer finds out that the application is nondeterministic. Columns 10 and 11 show the number of determinism checks in the application. There is a check at every *pthreads* barrier call, the end of the program, and, for `blackscholes` and `swaptions`, at the end of a loop iteration in a simulation pass. We do not check at hand-coded barriers, although such barriers can be as good points as *pthreads* barriers to expect determinism.

The first seven applications are bit-by-bit deterministic. Five of them use FP operations but are not affected by FP precision limitations because the parallelism does not trigger FP non-associative operations, illustrated in Figure 1(a). `volrend` has a benign data race in a hand-coded barrier, but InstantCheck correctly realizes that `volrend` is deterministic. `swaptions` is a Monte Carlo simulation, so one might expect it to be nondeterministic. However, `swaptions` uses thread-local random number generators that have no shared state. Thus, given the same seed, each thread generates a deterministic sequence of random numbers for itself, independent of the other threads or the thread interleavings.

`streamcluster` is an interesting case, because the original code we used (version 2.1) has a bug (a non-benign data race that creates an order violation) of which we were not aware before applying InstantCheck.[2] We found this bug only when investigating the nondeterminism reported by InstantCheck. For the *simmedium* input, the nondeterminism manifests at 74 dynamic barriers during the execution, after which it gets masked away and does not manifest at the end of the program. The nondeterminism created by this bug is not benign, because for small inputs (e.g., *simdev*), the nondeterminism propagates to the program's end and results in different outputs. This shows that checking determinism at as many points as possible during execution not only increases confidence in the program behavior but also catches bugs that for some inputs do not show up at the program end.

The next four applications (`fluidanimate`, `ocean`, `waterNS`, `waterSP`) are not bit-by-bit deterministic but they are deterministic modulo the FP precision limitations. Without InstantCheck's ability to ignore the small FP differences, these applications would appear to be highly nondeterministic, as illustrated for example in Figure 5(b).

Three applications (`cholesky`, `pbzip2`, `sphinx3`) are deterministic when ignoring small data structures. `cholesky` has three sources of nondeterminism: FP precision limitations, a nondeterministic custom memory allocator, and one nondeterministic data structure. We assume that the programmer wants to ignore the nondeterminism of the custom allocator, and we simply call *malloc* from inside the custom allocator. The nondeterministic data structure (in the field *freeTask*) is a per-thread singly linked list for free tasks. It is bit-by-bit nondeterministic because the order in which the tasks are linked, and the size of the list, differ from run to run. If we eliminate this linked list structure from the hash (as described in Section 2.2), `cholesky` turns out to be deterministic.

`pbzip2` has very high internal nondeterminism (many consumer threads race for jobs created by a producer), but `pbzip2` ends in a deterministic state if ignoring a pointer field in some result-task structures created by the consumers. The pointer field in these structures points to memory allocated nondeterministically by the consumers. The nondeterministic memory itself is deallocated during execution and thus no longer part of the program state, but the nondeterministic dangling pointers remain part of the program state. If we eliminate these pointers from the hash (Section 2.2), the program becomes externally deterministic. In addition, we hash in software the output stream that `pbzip2` writes to file, as described in Section 4.3. This stream is deterministic.

`sphinx3` is deterministic if ignoring about 4% of the memory state. The memory ignored is allocated at 15 out of the total 230 allocation sites in the code, which makes nondeterministic memory easy to identify and mark for deletion from the hash.

Three applications (`barnes`, `canneal`, `radiosity`) end up in nondeterministic states with many differences. Some nondeterministic code can be rewritten to be deterministic. If the algorithm permits, it is indeed better to implement deterministic code for easier development, testing, maintenance, and reuse. For example, a Java version of `barnes` was made deterministic in DPJ [6].

[2] After we reported the bug to the PARSEC author, he corrected it and informed us that the same bug had been previously reported internally. Our report was independent of the internal report.

Columns 10 and 11 list the number of dynamic points for which the program has deterministic/nondeterministic behavior: end of the program, barriers and, for `blackscholes` and `swaptions`, end of a loop iteration. These numbers confirm that even programs that were not written for determinism go through deterministic states, with no need (and no penalty) for enforcing internal determinism. Having many determinism points increases programmer's confidence in the code's behavior just like having a deterministic code is more reassuring than having a nondeterministic code. For example, the bug in `streamcluster` is detected only by 74 internal barriers, and checking at the remaining 12928 barriers would not catch the problem.

7.2.2 *Fast Detection of Nondeterminism*

We next discuss how quickly InstantCheck finds that an application is nondeterministic. Column 6 shows the first run when InstantCheck found bit-by-bit nondeterminism. In this set of experiments, it was the second or third run. This shows that the programmer finds out very quickly if the application is nondeterministic, which speeds up the testing process and increases programmer's productivity. It also means that InstantCheck does not require extra test runs over what programmers already test, since most testing strategies for parallel code run the code 10s to 1000s of times [8,35] for one given input, not just 2 or 3 times.

Since our experiments use a random scheduler, the numbers in Column 6 could change for a different seed for the scheduler, so it is important to consider not only the concrete numbers but their distribution across test runs. (This is a reason why we use 30 test runs for experiments even when nondeterminism is found in just 2 or 3 runs.) Figure 5 shows some of the nondeterminism distributions we observe in our experiments (the rest are omitted for brevity but are similar). These distributions show the number of different states observed when InstantCheck checks for determinism. For example, distribution *D_5* in Figure 5(c) means that, when running InstantCheck for `sphinx3`, there are 156 checking points (i.e., dynamic barriers) with the following behavior in the 30 test runs: 16 runs produce one state, 11 runs produce another state, and 3 runs produce yet another state. The same program has other groups of barriers with different distributions. A distribution of 29 and 1 would mean that 29 states are the same and only one differs, while a distribution of 30 means determinism for all 30 runs. As the distributions show, in practice, the probability of detecting nondeterminism is very high, and thus the nondeterminism detection in the second or third run as described earlier was not just by chance.

7.3 Negligible Overhead

We evaluate the performance of four configurations (Figure 6): (1) *Native* is the native application without checks for determinism, (2) *HW-InstantCheck*$_{Inc}$ is Native using HW-InstantCheck$_{Inc}$ to check for determinism, (3) *SW-InstantCheck*$_{Inc}$*-Ideal* is the lower bound for Native using SW-InstantCheck$_{Inc}$ to check for determinism, and (4) *SW-InstantCheck*$_{Tr}$*-Ideal* is the lower bound for the software-only traversal version, SW-InstantCheck$_{Tr}$. For the software-only versions we compute ideal lower bounds to show how fast highly-tuned implementations could get. We believe this makes the comparison with HW-InstantCheck$_{Inc}$ more fair than it would be by using our current prototypes of software-only versions, because these prototypes are not optimized for speed.

We use the *instruction count* as performance metric. For all configurations, we do not count the instructions of the randomizing thread scheduler. As explained in Section 7.1, the thread scheduler is not part of InstantCheck and in real usage, the scheduler would be replaced by the tool already used by the programmer

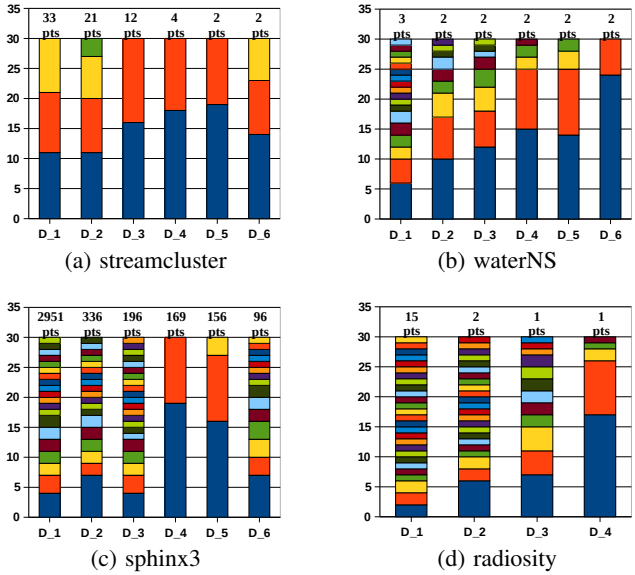


Figure 5: Distribution of nondeterminism points.

(e.g., PCT [8], CHESS [36], or stress test runner). We assume that our minor hardware addition has no significant impact on processor cycle time. HW-InstantCheck$_{Inc}$ does not incur additional cache misses in write-allocate caches (Section 3.1), but may increase read-port contention. We do not evaluate this contention, but believe the implementation choices described in Section 3.2 enable flexibility in scheduling most of the reads that may cause contention. We consider the cost for hashing one byte in software to be 5 instructions [20], and ignore the other costs for software-only versions (Section 4).

Figure 6 shows the number of instructions for all configurations, normalized to 1 for Native. HW-InstantCheck$_{Inc}$ incurs only a 0.3% overhead on average over the native runs, while the overhead for software versions ranges from 2% to 438X, with an average of 3X for SW-InstantCheck$_{Inc}$-Ideal and 5X for SW-InstantCheck$_{Tr}$-Ideal. HW-InstantCheck$_{Inc}$'s overhead is due to zeroing-out memory locations to prevent hash corruption (Section 5). For `sphinx3`, if the programmer wants to ignore the 4% of the memory state which is nondeterministic, both HW-InstantCheck$_{Inc}$ and SW-InstantCheck$_{Inc}$ delete this memory from the hash as described in Section 5. This operation creates 4.5X overhead for HW-InstantCheck$_{Inc}$ and 55X overhead for SW-InstantCheck$_{Inc}$-Ideal, which is still less than 438X for SW-InstantCheck$_{Tr}$-Ideal. HW-InstantCheck$_{Inc}$'s small overhead enables programmers to have determinism checking always-on to increase confidence in the developed software. In the absence of hardware support, programmers can use the software versions to check determinism when and as needed.

Figure 6 shows that, for some applications, one software version is clearly much faster than the other. For example, SW-InstantCheck$_{Inc}$-Ideal is much faster than SW-InstantCheck$_{Tr}$-Ideal for `ocean`, `sphinx3`, and `streamcluster`, but SW-InstantCheck$_{Tr}$-Ideal is faster for `barnes`, `fft`, and `lu`. SW-InstantCheck$_{Inc}$-Ideal is faster than SW-InstantCheck$_{Tr}$-Ideal when the number of writes between determinism checking points is small compared to the size of the program state, and thus updating the hash incrementally is cheaper than computing it by traversal. SW-InstantCheck$_{Tr}$-Ideal is faster when there are many writes between determinism checking points, and thus traversing the entire state is relatively cheap.

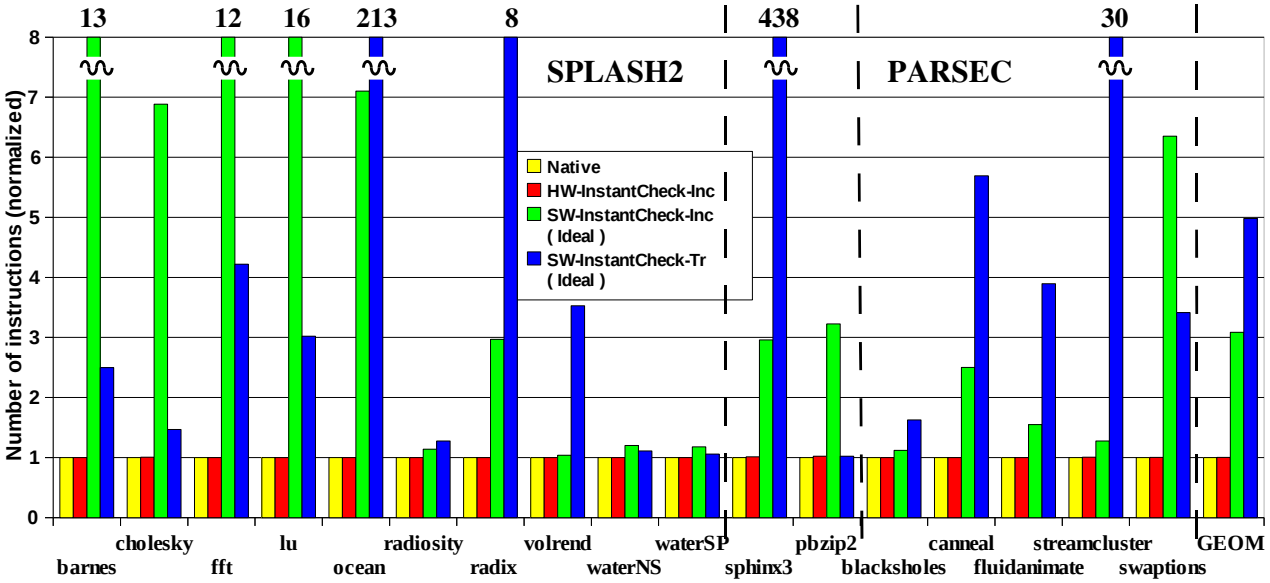


Figure 6: Number of instructions executed, normalized to Native. *GEOM* is the geometric mean.

kineti.C : KINETI()

```
LOCK(gl->KinetiSumLock);
if(ProcId == 3)
{    //        BUG
     SUM[dir]      =      S;
}
else
{    //        CORRECT
     SUM[dir]      +=     S;
}
UNLOCK(gl->KinetiSumLock);
```

(a)

interf.C : INTERF()

```
LOCK(gl->InterfVirLock);
if(ProcID==3){      //      BUG
     double tmp=*VIR;
     UNLOCK(gl->InterfVirLock);
     LOCK(gl->InterfVirLock);
     *VIR = tmp + LVIR/2.0;
}else{ //      CORRECT
     *VIR = *VIR + LVIR/2.0;
}
UNLOCK(gl->InterfVirLock);
```

(b)

radix.C : slave_sort()

```
if(MyNum == 3 && justOnce==3)
{    //      BUG
     ;      // no pause
}
else
{    //      CORRECT
     WAITPAUSE(n->done);
}
```

(c)

Figure 7: Seeded bugs. (a) semantic bug in `waterNS` (b) atomicity violation in `waterSP` (c) order violation in `radix`. We seed the bug only for thread 3. For `radix`, we also have only *one* dynamic occurrence (*justOnce==3*), since otherwise the program crashes.

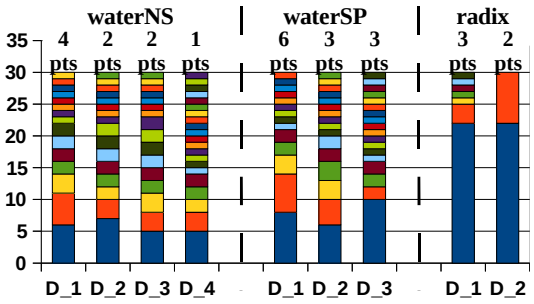


Figure 8: Distribution of nondeterminism points.

7.4 Bug Detection

InstantCheck focuses on determinism checking and not bug detection, but several projects [6, 9, 44] discuss how nondeterminism is related to bugs. Our own experience with the real bug that we found in `streamcluster` using InstantCheck (Section 7.2) encourages us to further investigate this aspect. We seed three bugs (semantic, atomicity violation, and order violation) in the applications from Section 7.2 as shown in Figure 7. The bugs do not cause program crashes but create incorrect results. To simulate rarely occurring bugs, we insert the buggy code path in only one thread, which decreases the number of buggy instructions executed. These bugs are of very different types (Section 2.3) and cannot be all detected by a technique that focuses on one particular bug type. For example, the semantic bug in `waterNS` would be very difficult to detect without an explicit verification of the numerical results at the end of the program. As another example, the atomicity violation in `waterSP` can be caught by an atomicity detector, but such tools can have fairly large runtime overheads and may require programmer annotations or extensive training runs [28].

The original applications are deterministic (under certain conditions, Table 1), and we want to find out if the bugs create nondeterminism that InstantCheck can detect. Table 2 shows the results. InstantCheck detects all three bugs with negligible overhead and without user annotations or training runs. InstantCheck also localizes the nondeterminism between two barriers, which helps in debugging (Section 2.3).

(Formerly) Deterministic Application	Bug Type	# Dynamic Checking Points		First NDet. Run
		Det	NDet	
waterNS	semantic	12	9	3
waterSP	atomicity violation	9	12	3
radix	order violation	7	5	6

Table 2: Bug detection results for 30 runs.

The last column of Table 2 shows the first run when the nondeterminism created by the bug is detected in our experiments. The bugs in `waterNS` and `waterSP` are detected very fast, in the third run. This fast detection is not just by chance, as confirmed by the scattered distributions for `waterNS` and `waterSP` in Figure 8. For `radix`, the bug is detected in the sixth run. Looking at the distribution for `radix` in Figure 8, we see that it is indeed less scattered than the ones for `waterNS` and `waterSP`.

8. RELATED WORK

Checking determinism properties can be performed dynamically or statically. DPJ [6] and SingleTrack [44] ensure *internal determinism* by imposing and checking (statically and dynamically, respectively) a particular programming style. Unlike DPJ and SingleTrack, InstantCheck does not impose any restrictions on how the code is written, as long as it results in a deterministic state. Any deterministic code written in DPJ or SingleTrack style is also detected as deterministic by InstantCheck. However, code that is detected as deterministic by InstantCheck may not be possible to express using DPJ or SingleTrack.

Burnim and Sen [9, 10] proposed dynamic checking of *semantic determinism*, which requires that selected data structures in the program behave in a certain, similar way across different runs. An example is that a set data structure always contains equal elements, although the order may change. The same approach was previously used in testing sequential code [51], where different runs come from executing different test sequences rather than the same sequence for different thread interleavings. Checking semantic determinism can handle arbitrary code, unlike DPJ and SingleTrack, but it can have high runtime overhead if large or many data structures are compared, and by default it checks neither internal nor external determinism because it requires explicitly specifying which locations to compare. External determinism by default compares entire states but allows explicitly specifying which locations to ignore. InstantCheck checks external determinism for entire program states with just 0.3% overhead with HW-InstantCheck$_{Inc}$.

Enforcing internally deterministic runs can be done each time code is executed, even for nondeterministic code, by approaches such as CoreDet [3], DMP [12], and Kendo [41]. Enforcing determinism at runtime does not impose a particular programming style but requires extensive hardware support [12], incurs significant overhead [3], or works only for race-free programs [41]. Unlike these proposals, InstantCheck has a minor hardware addition, small overhead, and is not affected by races. Also, InstantCheck checks if code is deterministic rather than making nondeterministic code execute deterministically.

Deterministic replay is a more researched area than deterministic execution. The classical approach [19, 33, 53] is to save a very precise execution log and use it to replay the exact thread interleaving as in the original run. Four recent proposals [1, 24, 25, 43] save only an imprecise log and use this log to replay a thread interleaving that is similar but not necessarily identical to the original execution. This is conceptually similar to having an externally deterministic program that has internal nondeterminism. Unlike these proposals, InstantCheck uses hashing to check program determinism and does not use an execution log to deterministically replay an original execution.

Incremental hashing is a powerful primitive that was used for several applications [15, 34, 39, 46]. In the architecture community, incremental hashing was used in the context of secure processors to avoid computing from scratch the node hashes of a hash tree [15] or to capture the set of memory reads and writes [46]. These are totally different applications of incremental hashing than the one of InstantCheck. In the software community, incremental hashing was used to speed up explicit-state model checking [34, 39]. Unlike InstantCheck, one technique [39] computes the hash of the state of the Promela model being checked and not the state of C/C++ parallel programs. The other technique [34] handles C/C++ programs but requires distinguishing pointers from primitive values to traverse the entire state in breadth-first order to detect isomorphic states that have the same relative shape of the heap even if the actual positions of objects differ.

Non-incremental hashing is also used for various tasks in software reliability. For example, PCC [7] and Light64 [40] compute hashes of calling chains and history of read values, respectively. In contrast, InstantCheck computes the hash of memory states and does so incrementally [2].

9. DISCUSSION

We believe hardware support for programmability should be extremely lightweight. *Hardware-based hashing* is one instance of such hardware: it is a powerful, versatile and extremely lightweight primitive that can improve a variety of programmability tasks. It is not unlike hardware Bloom filters [5], which were used for a variety of performance and programmability functions [11, 19, 32, 47, 54].

This paper uses hashing to capture the *state of a computation*. It uses on-the-fly incremental hashing to summarize the addresses and values written to memory. It demonstrates or outlines five uses of such hashing: checking for determinism, detecting software bugs, filtering out benign data races, speeding up systematic testing, and assisting deterministic replay.

Our previous work on Light64 [40] uses hardware hashing to capture the *history of a computation*. Light64 hashes loaded values and detects data races. PCC [7] uses (software) hashing to capture the *context of a computation* — it computes hashes of calling chains in serial Java programs — which can be also applied to several testing and debugging tasks.

Beyond this, we believe that hardware hashing has a largely-unexplored, broad design space of possibilities. Other information can be hashed, such as branch outcomes, and different hashing operations can be devised. Given all these possibilities, and the minimal hardware overhead required to support hashing, we hope to see many uses of this primitive for performance or programmability.

10. CONCLUSIONS

Showing that a program is externally deterministic has substantial benefits for software development and debugging. A deterministic program offers a higher reassurance of its correctness, will not produce unexpected outputs in a future run, and is much likelier to be reused. In this paper, we have presented *InstantCheck*, a novel technique that checks determinism with a very small runtime overhead. InstantCheck checks if the memory state produced by multiple runs of a parallel code is deterministic. On-the-fly incremental hashing enables a minimal and highly efficient hardware implementation: (1) the operations on the hashing register are core-local, and (2) the local operations on each hashing register can be performed in parallel and out-of-order. The hardware support can be used as a primitive for several other development and debugging tasks. Our use of InstantCheck showed that 14 out of 17 applications (`sphinx3`, `pbzip2`, PARSEC, and SPLASH-2) are deterministic when allowing for FP imprecision and small data structure differences. InstantCheck incurs a negligible average runtime overhead of 0.3% over native testing runs. Finally, InstantCheck helped us find a bug in the widely used PARSEC benchmark.

11. REFERENCES

[1] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[2] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Eurocrypt*, 1997.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 1970.

[6] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect sytem for deterministic parallel Java. In *OOPSLA*, 2009.

[7] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA*, 2007.

[8] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[9] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *ESEC/SIGSOFT FSE*, 2009.

[10] J. Burnim and K. Sen. DETERMIN: Inferring likely deterministic specifications of multithreaded programs. In *ICSE*, 2010.

[11] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *ISCA*, 2007.

[12] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[13] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[14] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, 2009.

[15] B. Gassend, G. E. Suh, D. E. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity. In *HPCA*, 2003.

[16] R. Ghiya, D. M. Lavery, and D. C. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *PLDI*, 2001.

[17] J. Gilchrist. PBZip2 v1.0.5. http://compression.ca/pbzip2.

[18] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rötteler. Using hardware transactional memory for data race detection. In *IPDPS*, 2009.

[19] D. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, 2008.

[20] B. Jenkins. A survey of hash functions. http://www.burtleburtle.net/bob/hash/doobs.html.

[21] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, 2005.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[23] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[24] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *MICRO*, 2009.

[25] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *ASPLOS*, 2010.

[26] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *IISWC*, 2005.

[27] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.

[28] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[29] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[30] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[31] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[32] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. G. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*, 2007.

[33] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *ASPLOS*, 2009.

[34] M. Musuvathi and D. L. Dill. An incremental heap canonicalization algorithm. In *SPIN*, 2005.

[35] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[36] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[37] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-based data race detection. In *ISCA*, 2009.

[38] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007.

[39] V. Y. Nguyen and T. C. Ruys. Incremental hashing for Spin. In *SPIN*, 2008.

[40] A. Nistor, D. Marinov, and J. Torrellas. Light64: Lightweight hardware support for data race detection during systematic testing of parallel programs. In *MICRO*, 2009.

[41] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[42] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[43] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[44] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *ESOP*, 2009.

[45] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[46] G. E. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *MICRO*, 2003.

[47] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. SoftSig: Software-exposed hardware signatures for code analysis and optimization. In *ASPLOS*, 2008.

[48] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *Automated Software Engineering Journal*, 10(2), 2003.

[49] W. Visser and P. Mehlitz. Model checking programs with Java PathFinder. In *ASE Tutorial*, 2006.

[50] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[51] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE*, 2004.

[52] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[53] M. Xu, M. D. Hill, and R. Bodík. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS*, 2006.

[54] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA*, 2007.

[55] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, 2010.