SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization

MANY CODE ANALYSIS TECHNIQUES FOR OPTIMIZATION, DEBUGGING, AND PARALLELIZATION MUST PERFORM RUNTIME DISAMBIGUATION OF ADDRESS SETS. HARDWARE SIGNATURES SUPPORT SUCH OPERATIONS EFFICIENTLY AND WITH LOW COMPLEXITY. SOFTSIG EXPOSES HARDWARE SIGNATURES TO SOFTWARE THROUGH INSTRUCTIONS THAT CONTROL WHICH ADDRESSES TO COLLECT AND WHICH TO DISAMBIGUATE AGAINST. THE MEMOISE ALGORITHM DEMONSTRATES SOFTSIG'S VERSATILITY BY DETECTING AND ELIMINATING REDUNDANT FUNCTION CALLS.

James Tuck
North Carolina State University
Wonsun Ahn
Josep Torrellas
University of Illinois, Urbana-Champaign
Luis Ceze
University of Washington

Many code-analysis techniques must ascertain at runtime whether two or more variables have the same address. Such runtime checks are the only choice when the compiler can't statically analyze the addresses. They provide crucial information that is used, for example, to perform various code optimizations, support breakpoints in debuggers, or parallelize sequential codes. Given the frequency and cost of performing these checks at runtime, researchers have proposed performing some of them in hardware.[1-3] Such proposals have different goals, such as ensuring that access reordering within a thread doesn't violate dependences, providing multiple hardware watchpoints for debugging, and detecting violations of interthread dependences in thread-level speculation (TLS). The expectation is that hardware-supported checking (or *disambiguation*) of addresses will have little time overhead.

A straightforward implementation of hardware-supported disambiguation can be complex and inefficient because it typically works by comparing an address to an associative structure with other addresses. For example, in TLS, when a processor writes, hardware matches its address against the addresses in other processors' speculative buffers (or caches). Similarly, intrathread access reordering checkers match a write's address against later reads that the compiler speculatively scheduled earlier. In general, longer speculation windows require larger associative structures. To improve efficiency, we could operate on sets of addresses simultaneously

Editor's Note:
© 2008 ACM, Inc. This is a minor revision of the work published in J. Tuck et al., *Proc. ASPLOS*; http://doi.acm.org/10.1145/1346281.1346300.

Published by the IEEE Computer Society

to compare many addresses in a single operation. Hardware signatures can accomplish this with low complexity.[4] In this case, we encode addresses using hash functions and accumulate them into a signature. If we provide hardware support for signature intersection as in Bulk,[4] address disambiguation becomes simple and fast.

Researchers have proposed signatures for address disambiguation in various situations, such as in load-store queues (LSQs)[5] and in TLS and transactional memory systems.[4,6,7] Typically, signatures are managed in hardware or have only a simple software interface.[6,7] However, to be truly useful for code-analysis and optimization techniques, signatures must provide a rich software interface.

To flexibly use signatures for advanced code analysis and optimization, we propose exposing a signature register file (SRF) to the software through a sophisticated instruction set architecture (ISA) called SoftSig. To demonstrate SoftSig's use, we propose an algorithm to efficiently detect redundant function calls and dynamically eliminate them. We call this memoization algorithm *Memoise*. Our results show that, on average for five popular multithreaded and sequential applications, Memoise reduces the number of dynamic instructions by 9.3 percent, thereby reducing the applications' average execution time by 9 percent. This article summarizes our work, which was presented in full elsewhere.[8]

Exposing signatures to software

An environment in which hardware signatures are flexibly manipulated in software must support three main operations: address collection, address disambiguation, and conflict detection. The software has a role in each of them.

In collection, the software specifies the window of program execution with memory accesses that must be recorded in a signature—that is, the set of program statements to be monitored, possibly with some restriction on the range of addresses to be recorded. Moreover, it specifies whether reads, writes, or both should be collected.

In disambiguation, the software specifies that the addresses collected in a given signature be compared to the dynamic stream of addresses accessed by the local thread, other threads (visible through coherence messages such as invalidations), or both. It also specifies whether reads and/or writes should be examined.

Finally, in conflict detection, the software specifies what action should be taken when the stream being monitored accesses an address in the signature. The action can be to set a bit that the software can later check or to trigger an exception and jump to a predefined location—possibly undoing the work performed in the meantime.

Examples

Figure 1 shows three examples of how we can use this environment: function memoization (Figures 1a and 1b), debugging with many watchpoints (Figure 1c), and loop-invariant code motion (LICM) (Figures 1d and 1e).

Function memoization involves dynamically skipping a call to a function if it can be proved that doing so won't affect the program state. Figure 1a shows two calls to function `foo` and some pointer accesses in between. Suppose the compiler can determine that the input argument's value is the same in both calls, but it can't prove whether the second call is dynamically redundant—due to nonanalyzable memory references inside or outside `foo`. With signatures (Figure 1b), the compiler allows address collection over the first call into a signature and then disambiguation of accesses against the signature until the next call. Before the second call, the code checks whether the signature observed a conflict. If it didn't, and no write in `foo` overwrites something read in `foo` (which can also be checked with signatures), the second invocation of `foo` can be skipped.

When debugging a program, it's desirable to know when a memory location is accessed. Debuggers offer this support in the form of a watch command, which takes as an argument an address to be watched, or *watchpoint*. Some processors provide hardware support to detect when a watchpoint is accessed.[9] However, because of the hardware costs involved, such processors support only a modest number of watchpoints (for example, four). Signatures let a processor watch numerous addresses simultaneously

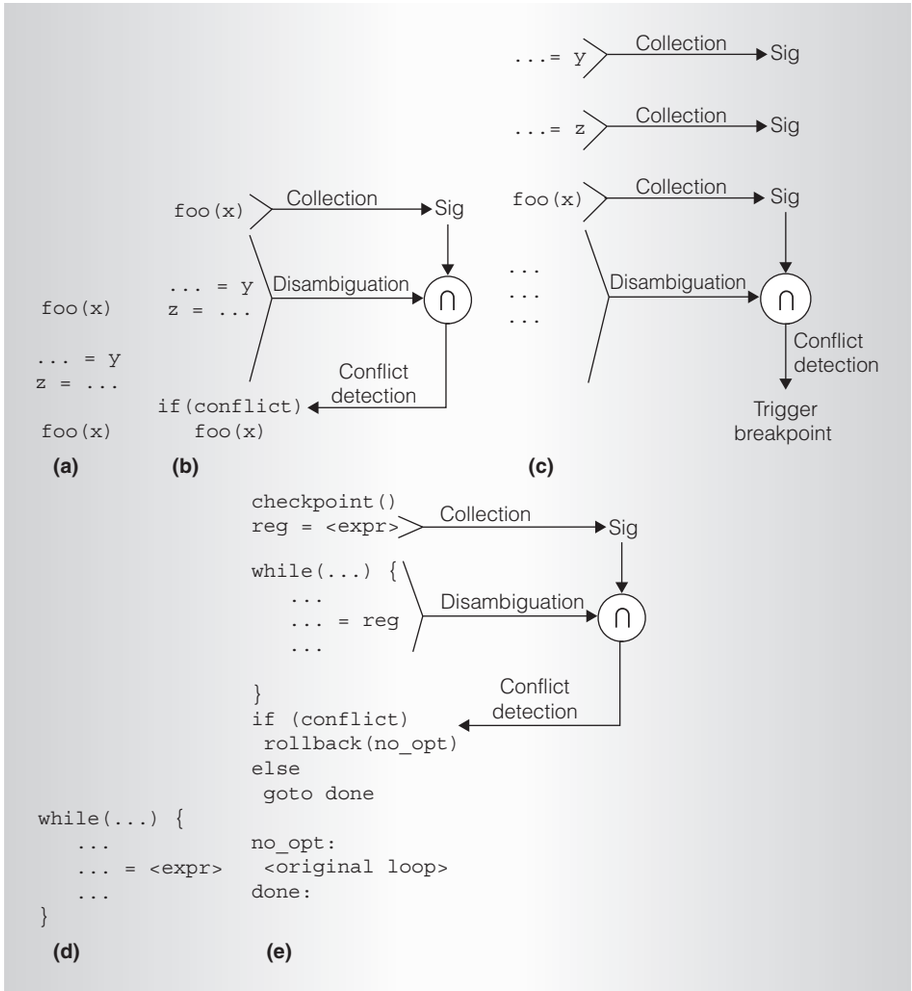

Figure 1. Three examples of using hardware signatures that are manipulatable in software: function memoization (a and b), debugging with many watchpoints (c), and loop-invariant code motion (LICM) (d and e).

with low overhead. For example, the code in Figure 1c collects addresses *y* and *z* in a signature. It then collects into the signature all the addresses that are accessed in `foo`. Next, it disambiguates all subsequent accesses against the signature, triggering a breakpoint if it detects a conflict. The system is watching for accesses to any of the addresses collected.

Finally, Figures 1d and 1e show a LICM example. Figure 1d shows a loop that computes an expression at every iteration. If the expression's value remains the same across iterations, moving the computation before the loop would offer savings. However, the code might contain nonanalyzable memory references that prevent the compiler from moving the code. With signatures and checkpointing support, the compiler can transform the code (see Figure 1e). Before the loop, hardware or software generates a checkpoint, and the program computes the expression and saves it in a register while collecting the addresses into a signature. Then, the loop is executed without the expression, while disambiguating against the signature. After the loop, the code checks whether the signature observed a conflict. If it did, the state is rolled back to the checkpoint and execution resumes at the beginning of the unmodified loop.

Design overview and guidelines

To expose hardware signatures to software, we extend a conventional superscalar processor with an SRF, which can hold a signature in each of its signature registers (SRs). Moreover, we add a few new instructions to manipulate signatures, enabling address collection, disambiguation, and conflict detection. The SoftSig architecture follows several design guidelines (labeled G1 to G5).

G1: Minimize SR accesses and copies. An SR is much larger than a 64-bit GPR—SoftSig SRs are 1 kilobit—and are therefore costly to read, move, and copy. Given an SR's size, it's important to minimize SR accesses and copies. Every move typically takes several cycles, and accessing the SRF consumes power. Consequently, we take several measures to minimize any negative impact on execution time or power consumption:

- On a context switch, the system doesn't save or restore SRs; rather, it discards signatures.
- The compiler never spills SRs to the stack.
- We design the logic to minimize reading SRs from the SRF.

Although these measures might appear to be severe limitations, our approach works well in spite of them.

G2: Manage the SRF through dynamic allocation. Because SRs are large, there are few of them. Moreover, to fulfill their use, SRs must be persistent—that is, once an SR begins collecting or disambiguating, it must remain in the SRF for the duration of the operation. Therefore, we must assign SRs so that we enable as many uses as possible in the program and use them where they're most profitable.

To maximize the number of uses, it's better to allocate the SRs dynamically than to reserve them based on static compiler analysis. For a given number of potential SR uses in a program, it might be difficult for the compiler to determine whether their lifetimes will overlap during execution. Consequently, the compiler might have to assume the worst case of maximum lifetime overlap and refrain from exploiting all opportunities. Dynamic allocation, on the other hand, uses dynamic information on the actual use lifetime to exploit as many opportunities at a time as SRs are available. This approach requires software routines or hardware logic to examine the SRF's current state and decide whether a SR can be allocated.

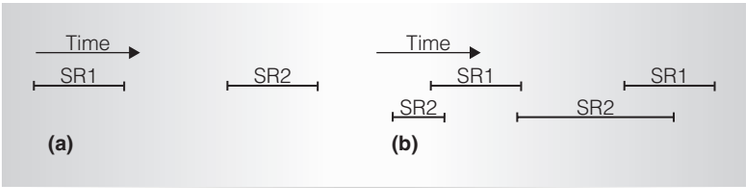

Figure 2. Employing SR1 and SR2 in uses with a lifetime (segment length) that is statically unpredictable: static allocation (a) and dynamic allocation (b).

Figure 2 shows two SRs and a program with four uses with hard-to-predict lifetimes. If we allocate SRs statically, we can only cover two uses. In practice, these two uses don't overlap in time (Figure 2a). If, instead, SRs are allocated dynamically, because at most two uses overlap in time, we can cover the four uses.

We leave the problem of deciding which SR uses are most worthwhile to the compiler, programmer, or a feedback-directed optimization framework.

G3: Ensure imprecision never compromises correctness. SoftSig must cope with multiple forms of imprecision—from imprecision in signature encoding, which can lead to false positives, to imprecision caused by the SRF's silent dynamic deallocation policy.

The system must be designed such that imprecision hurts at most performance and never correctness. Therefore, any software that uses an SR must be prepared to cope with a conflict that turns out to be a false positive. For instance, consider the watchpoint example in Figure 1c. A conflict might not be the result of an access to a watched location. The software must handle this case gracefully.

In addition, to handle unexpected deallocation of an in-use SR, SoftSig makes this event appear as if a conflict had occurred. Because the code must always work correctly in the presence of false positive conflicts, this approach will always be correct.

G4: Manage imprecision to provide the most efficiency. Because signatures might be silently

displaced while in use, an optimization can fail simply because of how the SRF is managed at runtime. To avoid such failure, we can manage some imprecision by controlling how SoftSig is used. Specifically, many false positives might indicate that SRs are too full and don't have enough precision. Using SRs over shorter code ranges or filtering some of the addresses are effective solutions to managing imprecision. SoftSig provides an instruction for filtering, which we describe later.

In addition, observing many SR deallocations indicates competing uses for the SRF. In this case, profiling can help determine the most profitable subset of SR uses. Software should judiciously manage both of these effects to provide the most efficiency.

G5: Minimize imprecision and unnecessary conflicts. Signatures will always have imprecision due to their hash-based implementation. However, to minimize additional sources of imprecision, the hardware must support address collection and disambiguation at precise instruction boundaries. Also, to minimize unnecessary conflicts, the system should perform disambiguation only against the addresses that are strictly necessary for correctness. This will reduce the number of unnecessary conflicts.

SoftSig software interface

Based on the previous discussion, SoftSig implements a detailed software interface for address collection and disambiguation, signature allocation and deallocation, and other operations on signatures. (Due to space limitations, we do not discuss the entire interface here; readers can find a detailed description elsewhere.[8])

Collection and disambiguation

Collection accumulates into an SR the addresses of the memory locations accessed during a window of execution. SoftSig supports collection using two instructions: `bcollect` and `ecollect`. When `bcollect` is executed, address collection begins. Depending on the instruction suffix, it will collect only reads (`rd`), only writes (`wr`), or both reads and writes (`r/w`). When `ecollect` is executed, address collection ends. Both instructions take as argument a general-purpose register (GPR) containing the SR's name.

Disambiguation checks for conflicts between addresses being accessed and a signature that has been or is currently being collected. SoftSig supports disambiguation using two instructions: `bdisamb` and `edisamb`. The former begins disambiguation, while the latter ends it. Both instructions take as argument a GPR that contains the SR's name. They demarcate a code region during which the hardware continually checks addresses for conflicts with the signature.

Disambiguation can be configured in many ways. One category of specification is whether the signature is disambiguated against accesses issued by the local processor (`bdisamb.loc` and `edisamb.loc`) or by remote ones (`bdisamb.rem` and `edisamb.rem`). Although the examples in Figure 2 all use local disambiguation, remote disambiguation is useful in a multithreaded program to identify when other threads issue accesses that conflict with those in a local signature. In addition, disambiguation can be configured to occur in only reads, only writes, or both reads and writes. As we'll see, remote disambiguation relies on the cache coherence protocol to flag accesses by remote processors. Consequently, signatures only observe those remote accesses that cause coherence actions in the local cache—for example, remote reads to a location that is only in shared state in the local cache won't be seen. In some cases, it might be desirable to disambiguate accesses performed by remote processors against a local signature that is currently being collected. In this case, we first need to use `bdisamb` to begin disambiguation and then `bcollect` to begin collection. Swapping the order of these two instructions is unsafe because it results in a window of time during which conflicts can be missed.

When disambiguation is enabled and the hardware detects a conflict with a signature, the hardware records it in a *status vector* associated with the signature. Later in this section, we show how the interface specifies the actions to take on a conflict.

For some optimizations, it is important to skip collection or disambiguation over a range of addresses that the compiler can guarantee need not be considered. This is

supported with the `filtersig` instruction. Its arguments are the SR's name and the beginning and end of the range—specified in registers using virtual addresses.

Persistence and signature status

The `allocsig` and `dallocsig` instructions allocate and deallocate, respectively, an SR in the SRF. Each instruction takes as an argument a GPR that holds the SR's name. The `allocsig` instruction further takes a second GPR that returns the SR's status vector. Finally, `allocsig` always allocates an SR, even if it requires silently displacing an existing signature. Consequently, any code optimization that uses SRs must be wary of the hardware displacing a signature it is relying upon.

The `sigstatv` instruction returns the status vector of an SR in a GPR. The status vector contains details about whether the signature is currently allocated, is zero, or has recorded a conflict. If `sigstatv` is called on a deallocated signature, a default status vector is returned that indicates it's no longer allocated and has a conflict. Consequently, all code optimizations must be implemented under the assumption that this default vector means that the signature can't be trusted to hold meaningful results.

Other operations on signatures

The interface supplies several other useful operations. It's possible to load, store, move, or clear a signature. Also, set operations such as insertion, intersection, and union are supported.

SoftSig architecture

The SoftSig architecture consists of several extensions to a superscalar processor. As Figure 3 shows, we group the extensions into a SoftSig processor module (SPM), which contains the SRF, status vectors, signature functional units (SFUs) to operate on signatures, exception vectors, and the in-flight conflict detector (ICD) module that aids remote disambiguation. The SPM interacts with the reorder buffer (ROB) and the LSQ to support collection and disambiguation.

SoftSig instruction execution

SoftSig instructions execute only when they reach the head of the ROB. Therefore, speculative instructions neither update nor allocate SRs. If we let speculative instructions update SRs, every speculative instruction that updated an SR would have to make a new copy of the SR to be able to support precise exceptions. The additional accesses and copies required would run counter to guideline G1. In addition, letting speculative instructions allocate SRs would induce more in-use SRs. This is at odds with guideline G2, which prescribes allocating and deallocating SRs dynamically in the most efficient manner.

Moreover, instructions do not update SRs out of order. Doing so would make it hard to maintain precise boundaries in the code sections where signatures are collected or disambiguated. The signatures would then be more imprecise, which would hurt guideline G5.

However, executing SoftSig instructions only when they reach the head of the ROB has two disadvantages. First, some non-SoftSig instructions might have data dependences with SoftSig instructions—for example, instructions that check the status vector. Such instructions must wait for the SoftSig instructions to execute. However, thanks to out-of-order execution, other, independent instructions can continue to execute. The second disadvantage is that we need the ICD module to ensure that remote disambiguation works correctly.

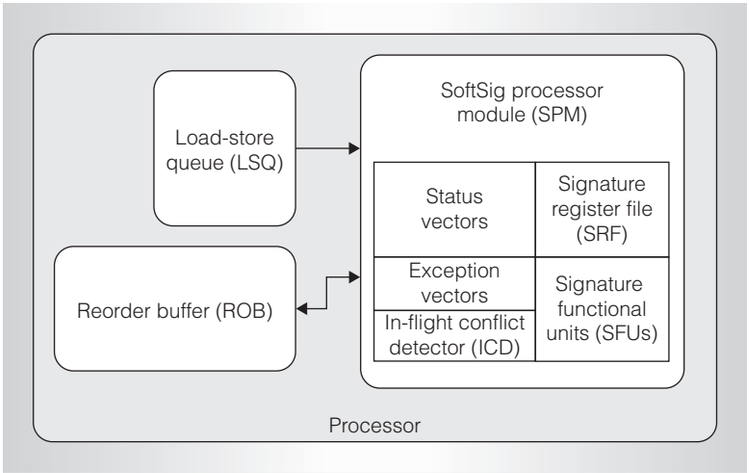

Figure 3. SoftSig architecture.

Signature register file

As Figure 4 shows, the SRF consists of three modules. The *signature register array* contains all the SRs and has a read (`Sig_Out`)

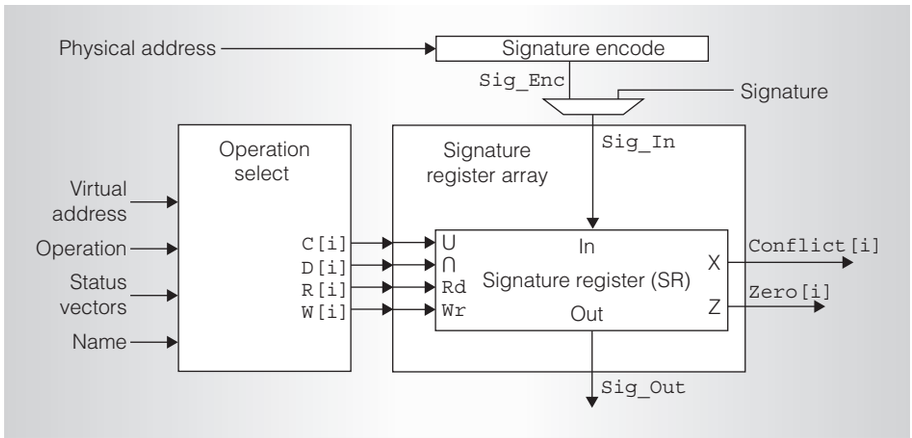

Figure 4. The signature register file (SRF).

and a write (`Sig_In`) port. Each SR has an input (`In`) and an output (`Out`) data port; control signals for union with the input (∪), intersection with the input (∩), read (`Rd`), and write (`Wr`); and output signals that flag a conflict (`Conflict[i]`) or a zero SR (`Zero[i]`).

The *operation select* module generates the control signals for the SRs. Specifically, it can set the collect (`C[i]`), disambiguate (`D[i]`), read (`R[i]`), or write (`W[i]`) signals for one or more SRs simultaneously. To generate these signals, it takes as inputs the status vectors of all the SRs and, if applicable, the type of operation to perform (`Op`), the name of the SR to operate on (`Name`), and the local access's virtual address (`VirtAddr`). We need the latter in case we need to filter ranges of addresses.

Finally, the *signature encode* module takes a physical address and transforms it into a signature (`Sig_Enc`). Either `Sig_Enc` or an explicit signature can be routed into the signature register array for collection, disambiguation, or writing.

Allocation and deallocation

When an `allocsig` instruction reaches the head of the ROB, the hardware attempts to allocate an SR. If an SR with the same name is already allocated, the system performs no action. Otherwise, the system clears an SR, initializes its status vector, and stores the SR name in the operation select module.

If no SR is free, the system selects one for displacement. It tries to displace an SR that has its conflict bit set. If no such SR exists, the system selects an SR randomly. In either case, it removes the deallocated SR's name from the operation select module.

When a `dallocsig` instruction reaches the head of the ROB, the hardware deallocates the corresponding SR. This operation involves removing the SR name from the operation select module.

Collection and local disambiguation

When a `bcollect` or `bdisamb.loc` instruction reaches the head of the ROB, the hardware notifies the LSQ to begin sending the address (virtual and physical) and access type of all memory operations as they retire to the SPM. In addition, it sets the appropriate bits in the corresponding status vector. As addresses are streamed into the SPM, the SRF handles them as we previously described.

If no conflict is detected on a memory operation, the hardware notifies the ROB that the corresponding instruction can retire; otherwise, it raises the conflict signal and, depending on the configuration, might generate an exception.

When an `ecollect` or an `edisamb.loc` instruction reaches the head of the ROB, the corresponding status vector is updated. When both collection and local disambiguation have terminated for all SRs, the LSQ no longer forwards state to the SPM.

Remote disambiguation

The `bdisamb.rem` instruction lets the SPM watch the addresses of external coherence actions, while the `edisamb.rem`

terminates this ability—if no other SR is performing remote disambiguation. Both instructions also update the corresponding SR's status vector. As usual, `edisamb.rem` performs its actions when it reaches the head of the ROB. However, `bdisamb.rem` is different in that, for correctness, it must perform some actions earlier.

Correctly supporting remote disambiguation

The challenging scenario occurs when SoftSig performs address collection and remote disambiguation on the same SR simultaneously. In this case, to eliminate any window of vulnerability in which SoftSig might miss a conflicting external coherence action, we enclose the `bcollect` and `ecollect` instructions inside the region bounded by `bdisamb.rem` and `edisamb.rem` instructions (see Figure 5a). Figure 5a also includes a load to variable *X* inside the code section being collected and remotely disambiguated.

However, as Figure 5b shows, out-of-order execution might cause the load to execute at time t_0, which is before it reaches the head of the ROB (and updates the SR) at time t_2. Unfortunately, if the processor receives an external invalidation on *X* at time t_1—between the time the load reads at t_0 and when it updates the SR at t_2—it will miss the conflict. We can't assume that the consistency model that the processor supports will force the retry of the load to *X*.

This inconsistency occurs because loads read data potentially much earlier than they update the SR. To solve this problem, we add the ICD, a counter-based bloom filter, to the SPM. The ICD tracks all inflight loads during remote disambiguation, and any external coherence action is disambiguated against both the SR and the ICD. If SoftSig finds a conflict on the ICD or the SR, it flags the SR with a conflict. In the example in Figure 5c, the ICD detects the conflict as soon as the invalidation is received. (Readers can find a detailed description of this mechanism elsewhere.[8])

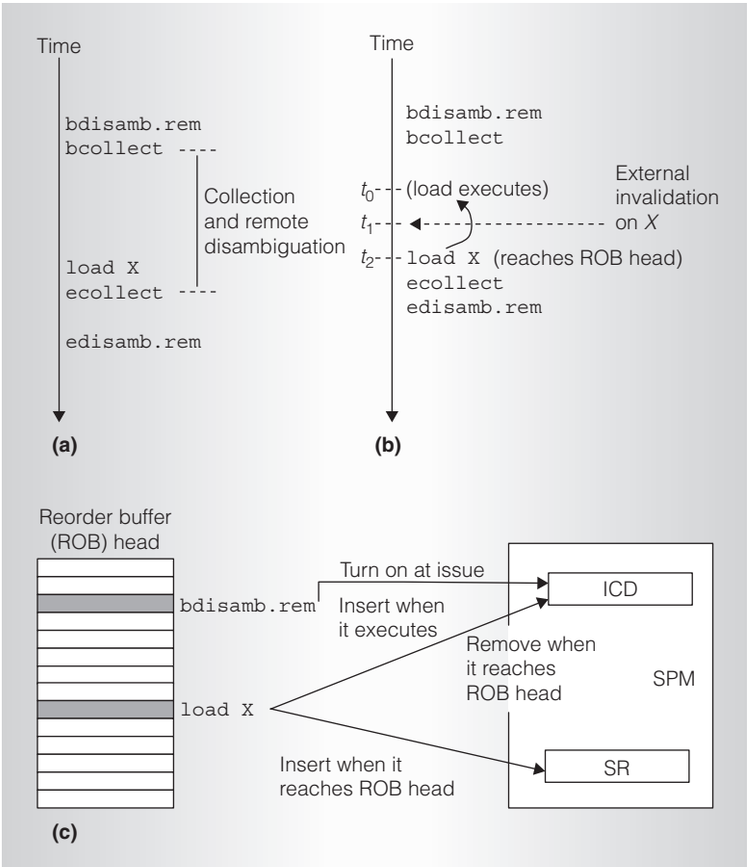

Figure 5. The in-flight conflict detector (ICD) prevents the system from missing a remote conflict: simultaneous collection and remote disambiguation (a), an invalidation missed due to out-of-order execution (b), and an invalidation observed because of the ICD (c).

Handling cache displacements under remote disambiguation

A final challenge to supporting remote disambiguation involves cache displacement. A cache is only guaranteed to see external coherence actions on those addresses that it caches. If the cache displaces a line, it might not see future coherence actions by other processors on that particular line. To prevent this, during remote disambiguation against an SR, the hardware takes a special action when a line is displaced from the cache. Specifically, it disambiguates the line's address against the SR, as if the cache had received an external invalidation on that line. This approach might conservatively generate a nonexisting conflict. However, it will never miss a real conflict.

Memoise: Signature-enhanced memoization

We propose Memoise, a general, low-overhead, and effective approach for function

memoization on C and C++ programs that exploits the SoftSig architecture.

In general, memoization algorithms cache the values of a function's inputs and outputs in a lookup table. When the function is next invoked, the algorithm searches the lookup table for an entry with an identical set of input values. If it finds such an entry, it copies the output values from the lookup table into the appropriate locations (memory or registers) and skips the function execution.

Unfortunately, a function's inputs and outputs aren't just the explicit input arguments passed to the function and the explicit output arguments it returns. They also include implicit inputs and outputs. These are other variables that the function reads from or writes to memory. With Memoise, we don't log implicit inputs and outputs. Instead, if none of the implicit inputs or outputs has been written to since the end of the function's previous invocation, they have the same values. We can easily check such a condition using SoftSig. It's possible that, during the function's execution, an implicit output overwrites a location read by an implicit input. In this case, because an input has changed, memoization should fail. Fortunately, detecting this case is easy with SoftSig.

Memoise algorithm

Memoise is implemented by intercepting function calls using code inserted in functions. Figure 6 shows the application of Memoise to function `foo`. Figure 6a shows the resulting layout of `foo`'s code. Memoise inserts three code fragments: *prologue*, *setup*, and *epilogue*. Figure 6b shows the statically allocated lookup table for `foo`. An entry in the table records the values of the explicit inputs and the explicit outputs of a call to `foo`. Different entries correspond to different values of the explicit inputs. In a multithreaded program, each thread has its own private copy of the lookup table to avoid needing to synchronize on access to a shared table.

Prologue. Figure 6c shows the prologue. It determines whether the call can be memoized and, if it can, reads out the explicit outputs stored in the lookup table and immediately jumps to the function return. To understand the code, note that each entry in the lookup table is logically associated with an SR. This SR collected the function's memory accesses when the function was called with the explicit inputs stored in the entry. Moreover, this SR was disambiguated against all local and remote accesses since that function call was executed. Finally, the name of this SR was set to be the lookup table entry's virtual address.

To test the memoization success, we check that the associated SR hasn't recorded a conflict since the function was last called with these explicit inputs. To perform the check, we use the `sigstatv` instruction to read the SR's status vector into register `R0`. If the bits in the status vector show that there has been no conflict and that this SR isn't currently collecting addresses (if it's still collecting, the function is recursive and, therefore, can't be memoized), memoization succeeded. In this case, the explicit outputs are read from the table entry and control transfers to the function return. Otherwise, the function is executed.

Setup. If the function call isn't memoized, the setup code fragment initializes the necessary structures to record this call's effects. Figure 6d shows the code, which involves three operations:

- obtaining a new entry in the lookup table (or recycling the entry that has the same explicit inputs, if it already exists),
- saving the explicit inputs in the entry, and
- starting up SRs to collect the implicit input and output addresses.

Figure 6d shows the instructions for the third operation. We allocate an SR each for addresses read and written. In the figure, the name of the SR for reads is the table entry's address and is stored in R1. We obtain the name of the SR for writes by adding 1 to the entry's address; it is stored in R2.

The next step is to skip the collection of (and the disambiguation against) the addresses of local accesses to memory-allocated variables that are neither implicit inputs nor implicit outputs. These are temporaries that are created on the stack for use during the call, or are explicit inputs or outputs passed on the stack. Figure 6f shows the stack locations where such variables

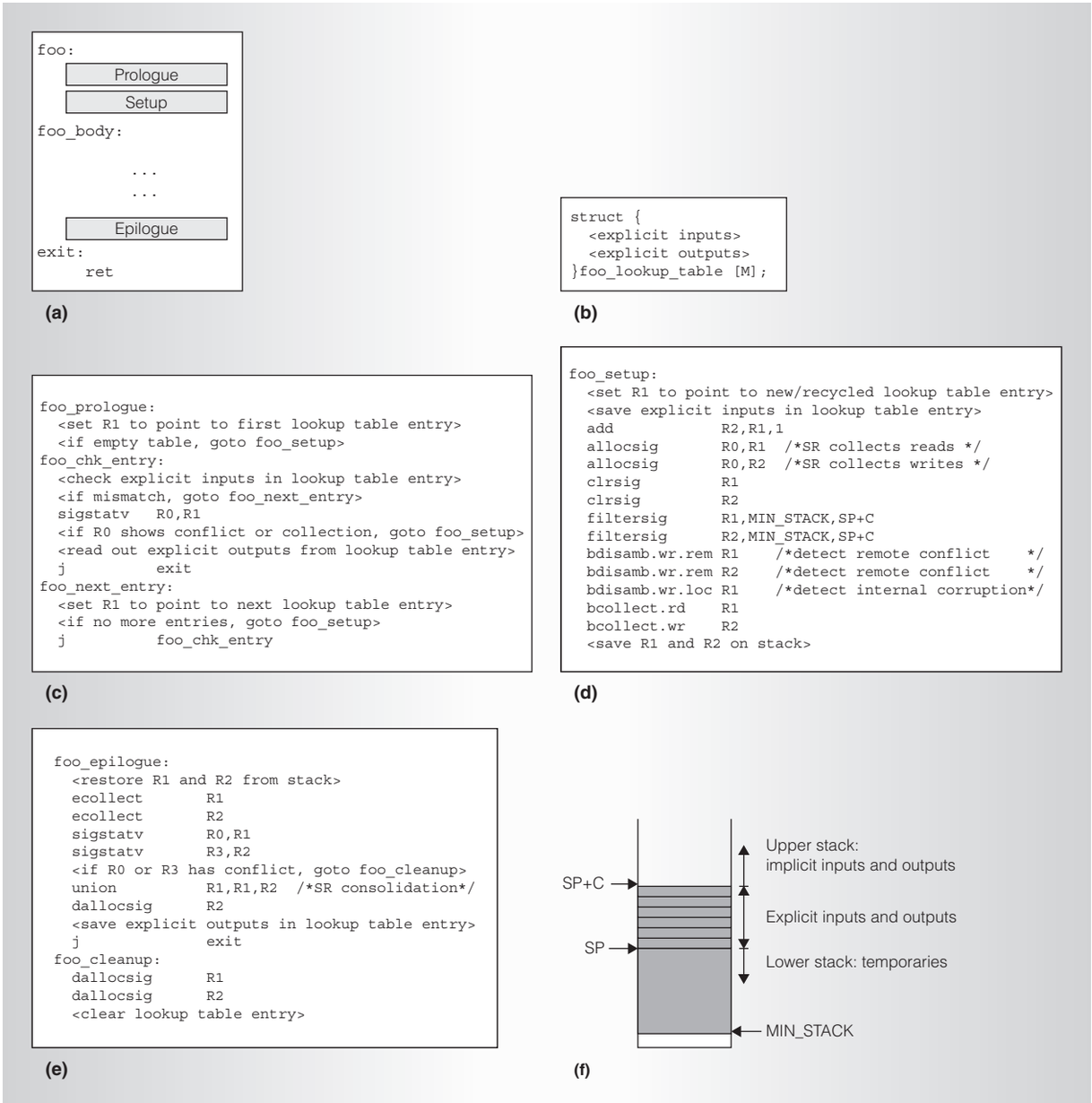

Figure 6. Applying the Memoise algorithm to function `foo`: function code layout (a), lookup table for `foo` (b), prologue (c), setup (d), epilogue (e), and stack layout (f).

are allocated. We store explicit inputs or outputs between SP and SP+C and store temporaries between MIN_STACK and SP. In Figure 6d, the `filtersig` instruction ensures that accesses to these variables are neither collected nor disambiguated against.

Next, we initiate disambiguation of remote writes against both SRs and of local writes against the SR that collects reads. The latter operation will detect internal corruption. Remote disambiguation is only necessary for multithreaded programs. Then, we start address collection for both SRs. Finally, we save R1 and R2 in the stack because we'll need them later.

Epilogue. After `foo` executes, we can fill the entry in the lookup table. The epilogue

Table 1. Environments analyzed.

Environment	Description
Baseline	No Memoise
Plain (P)	Memoise applied selectively to some functions using Ding and Li's cost-benefit analysis.[10] Lookup table size is limited to 10 entries.
Optimized (O)	P optimized by reducing the lookup table size to a single entry for functions that get little benefit from larger tables. It has low overhead.
Ideal (I)	O with unlimited number of SRs and no false positive conflicts. It approximates an ideal hardware behavior.

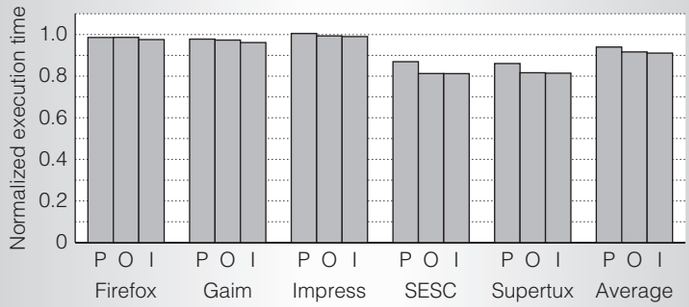

Figure 7. Execution times for three environments—plain (P), optimized (O), and ideal (I)—relative to baseline.

performs this process (see Figure 6e). This code first restores R1 and R2 from the stack and ends collection for both SRs. It then checks that they haven't recorded a conflict. If either has, memoization isn't possible, so we discard the entry in the lookup table and deallocate the two SRs.

Otherwise, we consolidate the two SRs into one SR (R1 in the example) to save space. Moreover, we save the call's explicit outputs in the lookup table entry. R1 remains under disambiguation against local and remote writes.

Optimizations for lower overhead

Because only some functions can benefit from memoization, we leverage an analytical model proposed by Ding and Li to identify which functions will most likely benefit from memoization.[10] Furthermore, searching a large lookup table usually adds significant overhead. We use a profiler to discover functions that mostly need a single-entry table and restructure the code to work more efficiently for this case.

Evaluation

To estimate Memoise's potential, we implemented an analysis tool that uses Pin,[11] a software framework for dynamic binary instrumentation. Pin's output is connected to a simulator of a multiprocessor memory subsystem based on the SESC superscalar simulator.[12] SESC models per-processor private L1 caches attached to a shared L2 cache. With this setup, we can estimate Memoise's reduction in the number of instructions executed and in execution time.

For our experiments, we ran several applications: Firefox, Gaim, Impress, SESC, and Supertux. The first three are popular applications used on many personal computers, and Supertux is an open source arcade game. Of these applications, Firefox, Impress, and Supertux are multithreaded, running with six, six, and two threads, respectively. For each application, we traced an execution of more than 400 million instructions.

We studied Memoise in the context of the four environments in Table 1. By default, we normalized the results to baseline.

Figure 7 shows the execution times of the plain, optimized, and ideal environments relative to baseline. On average, optimized offers a 9 percent reduction in execution time. Moreover, the reduction reaches 19 percent for SESC and Supertux. This is a significant reduction in execution time on challenging applications. In addition, the average reductions are nearly identical to those for the ideal environment and are better than for the plain environment. We also found that the execution time reductions closely follow the reductions in instruction count (as we describe elsewhere[8]).

SoftSig is useful for many other optimizations. We could revisit several proposals for runtime-disambiguation-based optimizations, with potentially new applications or more general use. Also, aggressive speculative code optimizations based on checkpointing might benefit from SoftSig's ability to record information about a program's dependences. Of course, SoftSig can integrate into environments that already

use signatures[4,6,7] to enhance the software's or the programmer's control over signature building and disambiguation. MICRO

References

1. D.M. Gallagher et al., ''Dynamic Memory Disambiguation Using the Memory Conflict Buffer,'' *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, 1994, pp. 183-193.
2. V. Krishnan and J. Torrellas, ''A Chip-Multiprocessor Architecture with Speculative Multithreading,'' *IEEE Trans. Computers,* vol. 48, no. 9, Sept. 1999, pp. 866-880.
3. G. Sohi, S. Breach, and T. Vijaykumar, ''Multiscalar Processors,'' *Proc. Int'l Symp. Computer Architecture,* ACM Press, 1995, pp. 414-425.
4. L. Ceze et al., ''Bulk Disambiguation of Speculative Threads in Multiprocessors,'' *Proc. Int'l Symp. Computer Architecture,* IEEE CS Press, 2006, pp. 227-238.
5. S. Sethumadhavan et al., ''Scalable Hardware Memory Disambiguation for High ILP Processors,'' *Proc. Int'l Symp. Microarchitecture,* IEEE CS Press, 2003, p. 399.
6. C. Minh et al., ''An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees,'' *Proc. Int'l Symp. Computer Architecture,* ACM Press, 2007, pp. 69-80.
7. L. Yen et al., ''LogTM-SE: Decoupling Hardware Transactional Memory from Caches,'' *Proc. Int'l Symp. High Performance Computer Architecture,* IEEE CS Press, 2007, pp. 261-272.
8. J. Tuck et al., ''SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization,'' *Proc Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), ACM Press, 2008, pp. 145-156.
9. Intel Corp., *Intel 64 and IA-32 Architectures Software Developer's Manual. Vol. 3B: System Programming Guide,* Part II, 2007.
10. Y. Ding and Z. Li, ''A Compiler Scheme for Reusing Intermediate Computation Results,'' *Proc. Int'l Symp. Code Generation and Optimization,* IEEE CS Press, 2004, p. 279.
11. C.-K. Luk et al., ''Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,'' *Proc. Int'l Conf. Programming Language Design and Implementation,* ACM Press, 2005, pp. 190-200.
12. J. Renau et al., ''SESC Simulator,'' 2005; http://sesc.sourceforge.net.

James Tuck is an assistant professor of electrical and computer engineering at North Carolina State University. His research interests include the interaction between compilers and architectures to create software with better performance, power, and reliability. Tuck has a PhD in computer science from the University of Illinois, Urbana-Champaign. He is a member of the IEEE and ACM.

Wonsun Ahn is a doctoral candidate in the Department of Computer Science at the University of Illinois, Urbana Champaign. His research interests include compiler-hardware hybrid optimization and hardware support for ease of parallel programming. Ahn has a BS in computer science and engineering from Seoul National University.

Luis Ceze is an assistant professor at the University of Washington. His research focuses on computer architecture and systems to improve the programmability of multiprocessor systems. Ceze has a PhD in computer science from the University of Illinois, Urbana-Champaign.

Josep Torrellas is a professor of computer science and Willett Faculty Scholar at the University of Illinois, Urbana-Champaign. His research interests include multiprocessor computer architecture and hardware and software reliability. Torrellas has a PhD in electrical engineering from Stanford University. He is an IEEE Fellow and a member of the ACM.

Direct questions and comments about this article to James Tuck, Center for Efficient, Secure, and Reliable Computing, North Carolina State Univ., Campus Box 7256, Raleigh, NC 27695-7256; jtuck@ncsu.edu.