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

James Tuck‡ Wonsun Ahn Luis Ceze† Josep Torrellas

‡NC State University
jtuck@ncsu.edu

University of Illinois at Urbana-Champaign
{dahn2, torrellas}@cs.uiuc.edu

†University of Washington
luisceze@cs.washington.edu

Abstract

Many code analysis techniques for optimization, debugging, or parallelization need to perform runtime disambiguation of sets of addresses. Such operations can be supported efficiently and with low complexity with hardware signatures.

To enable flexible use of signatures, this paper proposes to expose a Signature Register File to the software through a rich ISA. The software has great flexibility to decide, for each signature, which addresses to collect and which addresses to disambiguate against. We call this architecture *SoftSig*. In addition, as an example of SoftSig use, we show how to detect redundant function calls efficiently and eliminate them dynamically. We call this algorithm *MemoiSE*. On average for five popular applications, MemoiSE reduces the number of dynamic instructions by 9.3%, thereby reducing the execution time of the applications by 9%.

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: General; C.1.0 [*Processor Architectures*]: General

General Terms Performance, Design

Keywords Memory Disambiguation, Multi-core Architectures, Runtime Optimization

1. Introduction

Many code analysis techniques need to ascertain at runtime whether or not two or more variables have the same address. Such runtime checks are the only choice when the addresses cannot be statically analyzed by the compiler. 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, there have been many proposals to perform some of them in hardware (e.g., [9, 11, 12, 13, 22, 28]). Such proposals have different goals, such as ensuring that access reordering within a thread does not violate dependences, providing multiple hardware watchpoints for debugging, or detecting violations of inter-thread dependences in Thread-Level Speculation (TLS). The expectation is that hardware-supported checking (or "disambiguation") of addresses will have little overhead.

A straightforward implementation of hardware-supported disambiguation can be complex and inefficient. A key reason is that it typically works by comparing an address to an associative structure with other addresses. For example, in TLS, when a processor writes, its address is checked against the addresses in the speculative buffers (or caches) of other processors. Similarly, in intra-thread access reordering checkers (e.g., [9]), the address of a write is checked against later reads that have been speculatively scheduled earlier by the compiler. In general, longer windows of speculation require larger associative structures.

To improve efficiency, we would like to operate on sets of addresses at a time, so that, in a single operation, we compare many addresses. This can be accomplished with low complexity with hardware signatures [2]. In this case, addresses are encoded using hash functions and then accumulated into a signature. If we provide hardware support for signature intersection as in Bulk [3], then address disambiguation becomes simple and fast.

Signatures have been proposed for address disambiguation in various situations, such as in load-store queues (e.g., [25]) and in TLS and Transactional Memory (TM) systems (e.g., [3, 18, 31]). Typically, signatures are managed in hardware or have only a simple software interface [18, 31]. However, to be truly useful for code analysis and optimization techniques, signatures would need to provide a rich interface to the software.

To enable flexible use of signatures for advanced code analysis and optimization, this paper proposes to expose a Signature Register File to the software through a sophisticated ISA. The software has great flexibility to decide: (i) what stream of memory accesses to collect in each signature, (ii) what local or remote stream of memory accesses to disambiguate against each signature, and (iii) how to manipulate each signature. We call this architecture *SoftSig*, and describe the processor extensions needed to support it.

In addition, as an example of SoftSig use, this paper proposes an algorithm to detect redundant function calls efficiently and eliminate them dynamically. We call this memoization algorithm *MemoiSE*. Our results show that, on average for five popular multi-threaded and sequential applications, MemoiSE reduces the number of dynamic instructions by 9.3%, thereby reducing the average execution time of the applications by 9%.

This paper is organized as follows: Section 2 presents a background; Section 3 presents the SoftSig idea; Sections 4 and 5 present SoftSig's software interface and architecture; Section 6 describes MemoiSE; Section 7 evaluates MemoiSE; and Section 8 presents related work.

2. Background

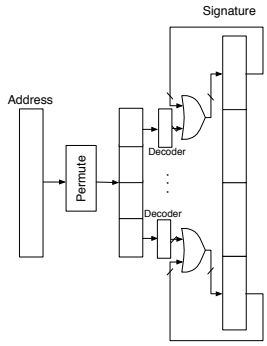
2.1 Hardware Signatures and Their Operations

Hardware signatures are special-purpose registers that store sets of addresses and can quickly disambiguate them. They are typically long — e.g., 1024 bits. A Bloom filter-based [2] hashing function

* This work was supported in part by the National Science Foundation under grants CHE-0121357 and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM, Intel, and Sun.

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.

like the one shown in Figure 1(a) generates a superset encoding of the addresses. Many addresses can be encoded into the fixed-length signature. Ceze *et al.* [3] proposed using hardware functional units that directly operate on signatures. Figure 1(b) shows some of the operations that can be performed on signatures. Intersection and union operations are performed via bit-wise AND and OR operations on the signatures. Interestingly, intersection can be interpreted as a disambiguation operation between two sets of addresses. Ceze *et al.* leveraged this insight to implement fast disambiguation in Hardware Transactional Memory (HTM) and TLS.



(a)

Op.	Description
$\cap$	Signature intersection
$\cup$	Signature union
$= \emptyset$	Is signature empty?
$\in$	Membership of an address in a signature

(b)

Figure 1. Address insertion into a signature (a) and some primitive operations on signatures (b).

Several other systems have later adopted hardware signatures and signature operations to speed up address disambiguation. For example, this includes proposals for TM [18, 31] and a proposal to enforce sequential consistency [4].

2.2 Memoization

Memoization is a technique that uses the basic observation that a function (or expression) that is called twice with the same inputs will compute the same result. Consequently, rather than computing the same result again, memoization involves storing the outcome in a lookup table and, on future occurrences of the function, simply returning the answer provided by the lookup table. Michie [17] first proposed memoization as a general way to avoid computing redundant work, and it is routinely applied in dynamic programming [7] and functional programming languages.

For memoization to be profitable, a function must be called with the same inputs often, to ensure a high hit rate in the lookup table. Also, the cost of the lookup must be less than that of executing the function. Not all potentially profitable functions can be memoized, however, since most functions in imperative languages like C have side effects or reads from nonlocal memory which are extremely hard to analyze statically [27]. In these cases, traditional memoization cannot be used.

3. Idea: Exposing Signatures to Software

3.1 Basic Idea

Many code analysis and optimization techniques, debugging schemes, and operations in speculative multithreading require the runtime disambiguation of multiple memory addresses — either accessed by a single thread or by multiple threads. We can significantly advance the art in these techniques if we support an environment where hardware signatures are flexibly manipulatable in software.

Such an environment must support three main operations: collection of addresses, disambiguation of addresses, and conflict detection. The software has a role in each of them. In *Collection*, the software specifies the window of program execution whose memory accesses must be recorded in a signature — i.e., 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 (i) the local thread, (ii) other threads (visible through coherence messages such as invalidations), or (iii) both. Again, 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 present 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.

3.2 Examples

Figure 2 shows three examples of how this environment can be used: function memoization (Charts (a) and (b)), debugging with many watchpoints (Chart (c)), and Loop Invariant Code Motion (LICM) (Charts (d) and (e)). Function memoization involves dynamically skipping a call to a function if it can be proved that doing so will not affect the program state. As an example, Figure 2(a) shows two calls to function `foo` and some pointer accesses in between. Suppose that the compiler can determine that the value of the input argument is the same in both calls, but is unable to prove whether or not the second call is dynamically redundant — due to non-analyzable memory references inside or outside `foo`. With signatures (Figure 2(b)), the compiler enables 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 if the signature observed a conflict. If it did not, and no write in `foo` overwrites something read in `foo`, then the second invocation of `foo` can be skipped.

A desirable operation when debugging a program is knowing 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 (e.g., [11]). However, due to the hardware costs involved, only a modest number of watchpoints is supported (e.g., 4). With signatures, a large number of addresses can be simultaneously watched with very low overhead. As an example, Figure 2(c) collects addresses y and z in a signature. Then, it collects into the signature all the addresses that are accessed in `foo`. After that, it disambiguates all subsequent accesses against the signature, triggering a breakpoint if a conflict is detected. The system is watching for accesses to any of the addresses collected.

Finally, Figures 2(d) and (e) show an example of LICM. Figure 2(d) shows a loop that computes an expression at every iteration. If the value of the expression remains the same across iterations, it would offer savings to move the computation before the loop. However, the code may contain non-analyzable memory references that prevent the compiler from moving the code. With signatures and checkpointing support, the compiler can transform the code as in Figure 2(e). Before the loop, a checkpoint is generated, and the expression is computed and saved 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 if 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.

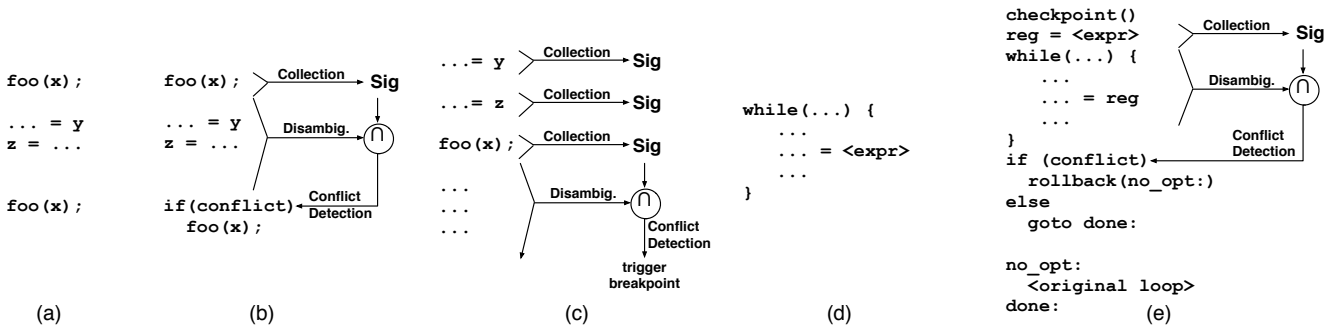


Figure 2. Three examples of how to use hardware signatures that are manipulatable in software.

3.3 Design Overview and Guidelines

To expose hardware signatures to software, we extend a conventional superscalar processor with a *Signature Register File* (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. We call our architecture *SoftSig*. Before describing SoftSig, we outline some design guidelines that we follow. The guidelines are listed in Table 1.

G1	Minimize SR accesses and copies
G2	Manage the SRF through dynamic allocation
G3	Imprecision should never compromise correctness
G4	Manage imprecision to provide the most efficiency
G5	Minimize imprecision and unnecessary conflicts

Table 1. Design guidelines in SoftSig.

3.3.1 Signature Registers are Unlike General Purpose Registers

SRs must be treated differently than General Purpose Registers (GPRs) because they are different in two ways. First, an SR is much larger than a 64-bit GPR — SRs are 1 kilobit in SoftSig. Due to their size, SRs are costly to read, move and copy. Second, SRs are persistent. Once a SR begins collecting or disambiguating, it must remain in the SRF for the duration of the operation in order to work as expected. An operation may take a very long time to complete, as can be seen from the examples in Figure 2.

These observations motivate two design guidelines:

G1*: Minimize SR accesses and copies.* Given the size of SRs, it is important to minimize SR accesses and copies. Every move typically takes several cycles, while accessing the SRF consumes power. Consequently, we minimize any negative impact on execution time or power consumption through several measures. First, on a context switch, the system does not save or restore SRs; rather, signatures are discarded. Second, the compiler never spills SRs to the stack. Finally, we design the logic to minimize reading SRs from the SRF. While these measures may appear to be severe limitations, our approach works well in spite of them.

G2*: Manage the SRF through dynamic allocation.* Given the size of SRs, there are few of them. Moreover, given their persistence, their use must be coordinated across an entire program's execution. This introduces the issue of how to assign SRs so that (i) we enable as many uses as possible in the program and (ii) we use them where they are most profitable.

To maximize the number of uses, it is better to allocate the SRs dynamically than to reserve the SRs based on static compiler analysis. For a given number of potential SR uses in a program, it may be difficult for the compiler to determine whether or not the lifetimes of these uses will overlap in time during execution. Consequently, the compiler may 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 current state of the SRF and decide whether a SR can be allocated.

As an example, Figure 3 shows the case of 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 do not overlap in time (Chart (a)). If, instead, SRs are allocated dynamically, since at most two uses overlap in time, we can cover the four uses.

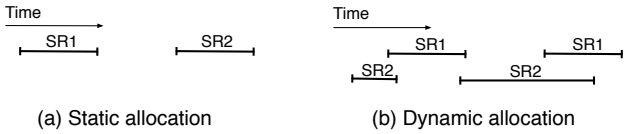


Figure 3. Employing SR1 and SR2 in uses whose lifetime (length of the segment) is unpredictable statically.

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

3.3.2 System Must Cope with Imprecision

SoftSig must cope with multiple forms of imprecision. One form of imprecision is the encoding of signatures. Instead of an exact list of addresses, only a superset of addresses is actually known. Because of this, conflicts may be reported even when an exact list would show that there were none. Such conflicts are called *false positives*.

Another source of imprecision is the dynamic de/allocation policy of the SRF. Signatures may be silently displaced while in use. Consequently, an optimization can fail simply because of how the SRF is managed at runtime.

The presence of imprecision motivates three design guidelines:

G3*: Imprecision should never compromise correctness.* 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 2(c). A conflict may not be the result of an access to a watched location. The software needs to handle this case gracefully.

In addition, to handle the case of the unexpected deallocation of an in-use SR, SoftSig makes this event appear as if a conflict had occurred. Since the code must always work correctly in the

Category	Instruction	Description
Collection	bcollect.(rd,wr,r/w) R1	Begin collecting addresses into SRF[R1]. Depending on the specifier, collect only reads, writes, or both
	ecollect R1	End collecting addresses into SRF[R1]
	filtersig R1,R2,R3	Do not collect or disambiguate addresses between R2 and R3 into/against SRF[R1]
Disambiguation	bdisamb.(rd,wr,r/w).(loc,rem) R1	Begin disambiguating local or remote accesses (depending on the specifier) against SRF[R1]. Depending on the specifier, disambiguate only reads, writes, or both
	edisamb.(loc,rem) R1	End disambiguating local or remote accesses (depending on the specifier) against SRF[R1]
Persistence, Status, & Exceptions	allocsig R1,R2	Allocate register SRF[R2] and return its Status Vector in R1
	dallocsig R1	Deallocate SRF[R1]
	sigstatv R1,R2	Return the Status Vector of SRF[R2] in R1
	exptsig R1,*target*	Except to *target* if a conflict occurs on SRF[R1]
Manipulation	ldsig R1,*addr*	Load from *addr* into SRF[R1]
	stsig R1,*addr*	Store SRF[R1] to *addr*
	mvsig R1,R2	SRF[R1] $\leftarrow$ SRF[R2]
	clrsig R1	SRF[R1] $\leftarrow \emptyset$, clear Status Vector and set a=1,z=1,x=0
	union R1,R2,R3	SRF[R1] $\leftarrow$ SRF[R2]$\cup$ SRF[R3]
	insert_elem R1,R2	SRF[R2] $\leftarrow$ R1 $\cup$ SRF[R2]
	member R1,R2,R3	R1 $\leftarrow$ (R2 $\in$ SRF[R3]) ? 1 : 0
	intersect R1,R2,R3	SRF[R1] $\leftarrow$ SRF[R2] $\cap$ SRF[R3]

Table 2. SoftSig software interface.

presence of false positive conflicts, this approach will always be correct.

G4*: Manage imprecision to provide the most efficiency.* Some imprecision can be managed in software by controlling how SoftSig is used. Specifically, many false positives may indicate that SRs are too full and do not have enough precision. Using SRs over shorter code ranges or finding a way to filter some of the addresses are effective solutions to manage imprecision. SoftSig provides an instruction for filtering (Section 4).

In addition, many SR deallocations indicate competing uses for the SRF. In this case, profiling can help determine the subset of SR uses that are most profitable. 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, disambiguation should be performed only against the addresses that are strictly necessary for correctness. In so doing, the number of unnecessary conflicts will decrease.

4. SoftSig Software Interface

Based on the previous discussion, this section describes SoftSig's software interface.

4.1 The Signature Register File (SRF)

A core includes an SRF, which holds a set of SRs. SRs are not saved and restored at function calls and returns. Instead, SRs have persistence — they are allocated when needed and, in normal circumstances, deallocated only when they are not needed anymore. Consequently, when an SR is allocated, it is assigned a *Name* specified by the program. Such a name is used to refer to it until deallocation. The instructions used to manipulate SRs constitute SoftSig's software interface. They are shown in Table 2 and discussed next.

4.2 Collection

Collection is the operation that accumulates into an SR the addresses of the memory locations accessed during a window of execution. SoftSig supports collection using two instructions, namely `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) that contains the name of the SR.

For some optimizations, it is important to skip collection over a range of addresses that the compiler can guarantee need not be considered. This is supported with the `filtersig` instruction. Its inputs are the name of the SR, and the beginning and end of the range — specified using virtual addresses.

4.3 Disambiguation

Disambiguation is the operation that checks for conflicts between addresses being accessed and a signature that has been collected or is currently being collected. SoftSig supports disambiguation using two instructions, namely `bdisamb` and `edisamb`. The former begins disambiguation, while the latter ends it. Both instructions take as argument a GPR that contains the name of the SR. 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 or by remote ones. While the examples in Figure 2 all used 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 will 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 — e.g., remote reads to a location that is only in shared state in the local cache will not be seen.

In some cases, it may be desirable to disambiguate accesses performed by remote processors against a local signature that is currently being collected. This often occurs under HTM or TLS. 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 when 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 will show how the interface specifies the actions to take on a conflict.

The `filtersig` instruction blocks disambiguation over its range of addresses for the specified signature.

4.4 Persistence, Signature Status, and Exceptions

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 name of the SR. `allocsig` further takes a second GPR that returns the Status Vector of the SR. 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. Figure 4 shows the fields of the vector. For now, consider the three 1-bit fields in the figure that are unshaded. They describe whether the signature is currently allocated (*a*), is zero (*z*), or has recorded a conflict (*x*). If `sigstatv` is called on a deallocated signature, a default Status Vector is returned with *a*=0, *z*=0, and *x*=1. Consequently, all code optimizations have to be implemented under the assumption that this default vector means that the signature cannot be trusted to hold meaningful results.

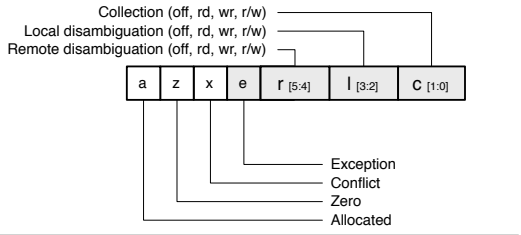


Figure 4. Status Vector associated with a signature.

While this makes it possible to generate code that will always function correctly, it would be inefficient to require a Status Vector check before every signature operation. Therefore, SoftSig supplies an additional simplifying policy: a disambiguation or collection operation on a deallocated signature is converted into a NOP.

The `sigstatv` instruction makes it possible to explicitly query for the presence of a conflict. However, it is not always desirable to schedule an instruction to test for a conflict. Consider the case of watchpoints in Figure 2(c) — any conflict should be reported immediately when it occurs. To enable such behavior, SoftSig provides the `exptsig` instruction. `exptsig` specifies an exception handler that should be triggered when a conflict occurs on a specific SR. `Exptsig` takes as arguments the SR and the address of the first instruction of the exception handler.

The shaded fields in the Status Vector shown in Figure 4 supply additional configuration information that is only valid if the signature is allocated. Starting from the right, the fields show the status of collection, and of local and remote disambiguation. The status can be off, only reads, only writes, and reads plus writes. The *e* bit indicates whether an exception should be generated on a conflict.

4.5 Signature Manipulation

SoftSig provides a set of operations to manipulate signatures. Table 2 lists them. Since they are straightforward, we leave it to the reader to understand their use from the description in the table.

4.6 Interaction with Checkpointing

As shown Figures 2(a)-(c), SoftSig is useful without the need for machine checkpoints. However, if the system supports checkpointing — either in software or in hardware — SoftSig can provide additional functionality. Specifically, it can enable efficient execution of optimizations where the code performs some risky operation speculatively and then tests whether the execution was correct. If it was not, execution is rolled back.

Figures 2(d)-(e) showed an example of speculative optimization. The expression is assumed loop invariant and hoisted before the loop. After the loop is executed, there is a check to see if the assumption was correct. If it was not, the checkpoint is restored and the original loop is executed. Note, however, that SoftSig's applicability is not limited to speculative environments.

4.7 Managing Signature Registers

It is necessary that each SR have a unique name. If two SRs had the same name, they could be confused with one another and lead to incorrect programs. Within a thread, the compiler can typically guarantee that each dynamic SR instance has a different name. For example, it can derive the name at allocation time based on the address of the function that allocates the SR.

This approach, however, does not guarantee that names are unique across different threads or processes time-sharing a processor. One possible solution is to include the thread or process ID as part of the name of the SR. This approach works well for SMT processors. For single-threaded processors, by simply invalidating the SRF at context switches as per guideline **G1**, we eliminate any possible confusion between SR instances.

5. SoftSig Architecture

The SoftSig architecture consists of several extensions to a superscalar processor. As shown in Figure 5, the extensions are grouped into a *SoftSig Processor Module* (SPM), which contains the Signature Register File (SRF), the Status Vectors, FUs to operate on signatures, the exception vectors, and a module called the In-flight Conflict Detector (ICD), which aids remote disambiguation. The SPM interacts with the Reorder Buffer (ROB) and the Load-Store Queue (LSQ) in support of collection and disambiguation. The rest of this section describes the architecture in detail.

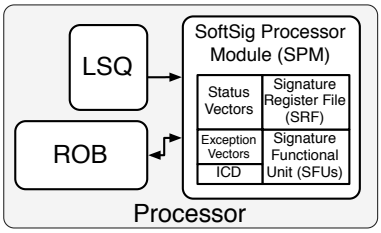


Figure 5. SoftSig architecture.

5.1 SoftSig Instruction Execution

In our design, SoftSig instructions execute only when they reach the head of the ROB. Therefore, SRs are neither renamed nor updated by speculative instructions or updated out of order. We choose this approach to follow guidelines **G1**, **G2**, and **G5** in Section 3.3. Indeed, if we allowed speculative instructions to update SRs, every speculative instruction that updated an SR would have to make a new copy of the SR, in order to be able to support precise exceptions. The additional accesses and copies required would run counter to guideline **G1**.

In addition, allowing speculative instructions to update SRs would induce a larger number of in-use SRs. This is at odds with guideline **G2**, which prescribes that the SRs should be allocated and deallocated dynamically in the most efficient manner. Finally, allowing out-of-order update of the SRs 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 may have data dependences with SoftSig instructions — for example, instructions that check the Status Vector. Such instructions will have to wait for the SoftSig ones to execute. However, thanks to out-of-order execution, other, independent instructions can continue to execute. The second disadvantage is that remote disambiguation does not work correctly in this environment unless the ICD module is added. Section 5.5.1 presents this problem in more detail and describes our solution.

5.2 Signature Register File

As shown in Figure 6, the SRF is composed of three modules, namely the Signature Register Array, the Operation Select, and the Signature Encode. The former contains all the SRs, and has a read (*Sig_Out*) and a write (*Sig_In*) port. In turn, 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]*).

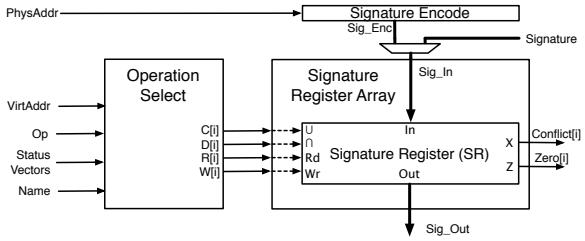


Figure 6. The signature register file.

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 virtual address of the local access (*VirtAddr*). The latter is needed 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.

5.3 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, no action is performed. Otherwise, an SR is cleared, its Status Vector is initialized, and the SR name is stored in the Operation Select module.

If there is no free SR, then one is selected for displacement. The system tries to displace an SR that has its Conflict bit set. If no such SR exists, then an SR is selected at random. In either case, the name of the deallocated SR is removed 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.

5.4 Collection and Local Disambiguation

When a `bcollect` or a `bdisamb.loc` instruction reaches the head of the ROB, the hardware notifies the LSQ to begin sending to the SoftSig Processor Module (SPM) the address (virtual and physical) and type of access of all memory operations as they retire. In addition, the appropriate bits in the corresponding Status Vector are set. As addresses are streamed into the SPM, they are handled by the SRF as described previously.

If no conflict is detected on a memory operation, the ROB is notified that the corresponding instruction can retire; otherwise, the Conflict signal is raised and, depending on the configuration, an exception may be generated (Section 5.6).

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 does not forward state to the SPM any longer.

5.5 Remote Disambiguation

The `bdisamb.rem` instruction enables the SPM to 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 Status Vector of the corresponding SR. 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 needs to perform some of its actions earlier. In the following, we consider why this is the case and how we ensure correct remote disambiguation.

5.5.1 Correctly Supporting Remote Disambiguation

The challenging scenario occurs when SoftSig performs address collection and remote disambiguation on the same SR simultaneously. This is a common situation in HTMs. In this case, to eliminate any window of vulnerability where a conflicting external coherence action could be missed, we must enclose the `bcollect` and `ecollect` instructions inside the region bounded by `bdisamb.rem` and `edisamb.rem` instructions. This is shown in Figure 7(a), which also includes a load to variable *X* inside the code section being collected and remotely disambiguated.

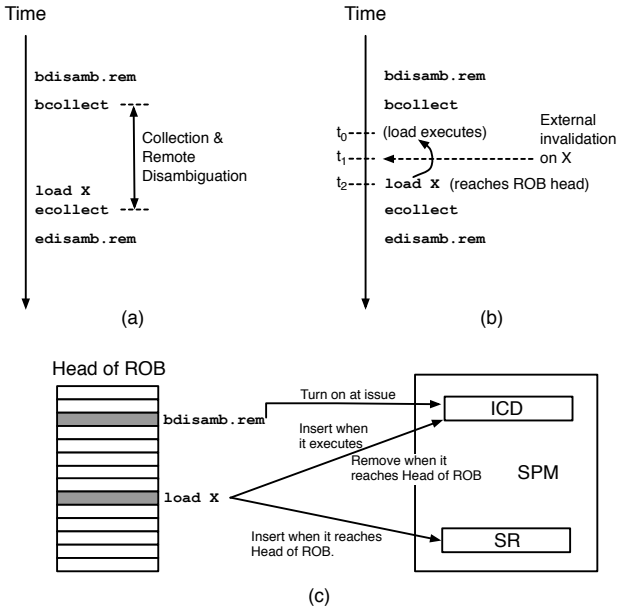


Figure 7. The ICD prevents missing a remote conflict.

However, as shown in Figure 7(b), due to out-of-order execution, the load may 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 an external invalidation on *X* is received at time t_1 — in between the time the load reads at t_0 and the time it updates the SR at t_2 — the conflict will be missed. Note that we cannot assume that the

consistency model supported by the processor 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 *In-flight Conflict Detector* (ICD) to the SPM, and require that `bdisamb.rem` perform most of its actions in the (in-order) issue stage. More specifically, the ICD is a counter-based Bloom filter that automatically accumulates the addresses of all *in-flight* loads. As shown in Figure 7(c), when the load executes, *X* is inserted into the ICD. When the load reaches the head of the ROB, *X* is inserted into the SR and, since the ICD is counter-based, it is removed from the ICD. If remote address disambiguation is enabled, any external coherence action is disambiguated against both the SR and the ICD. If a conflict is found on either the ICD or the SR, then the SR is flagged as having a conflict. In the example shown, the conflict will be detected by the ICD as soon as the invalidation is received.

Moreover, we must ensure that `bdisamb.rem` turns the ICD on before any subsequent load could be executed. Consequently, we conservatively require that `bdisamb.rem` turn the ICD on and start directing external coherence addresses to the SPM as soon as the `bdisamb.rem` instruction goes through the (in-order) issue stage. This is shown in Figure 7(c). However, `bdisamb.rem` does not update the Status Vector until it reaches the head of the ROB. This is because only then can we guarantee that the corresponding `allocsig` instruction has retired.

Based on this design, the full behavior of the ICD is as follows. When `bdisamb.rem` is issued, the ICD is turned on. When a load executes, its address is added to the ICD; when a load reaches the head of the ROB or is found to be misspeculated, its address is removed from the ICD. If an external coherence action has a conflict with the ICD, the ICD sets a flag indicating a conflict and remembers the ROB index of the youngest load instruction *i* that has executed so far. All SRs that are collecting and performing remote disambiguation from this point until *i* retires will have their Conflict bit set in their Status Vector. Once *i* retires, the ICD clears its conflict flag — since any SR that starts collection after *i* should not be affected by this conflict. The ICD remains active from the time the first `bdisamb.rem` is issued until no signatures perform remote disambiguation anymore.

5.5.2 Handling Cache Displacements under Remote Disambiguation

A final challenge to supporting remote disambiguation involves cache displacements. The problem is that a cache is only guaranteed to see external coherence actions on those addresses that it caches. If the cache displaces a line, the cache may not see future coherence actions by other processors on that particular line. Therefore, consider a processor that performs both collection and remote disambiguation on an SR. Suppose that the processor references a line, inserts its address in the SR, and then displaces the line from the cache. Future coherence actions by other processors on that line may not be seen by the cache and, therefore, remote disambiguation cannot be trusted to identify all remote conflicts.

To prevent this case, when remote disambiguation against an SR is in progress, the hardware takes a special action when a line is displaced from the cache. Specifically, the line's address is disambiguated against the SR, as if the cache had received an external invalidation on that line. This approach may conservatively generate a non-existing conflict. However, it will never result in missing a real conflict.

A more expensive, alternative approach would involve explicitly preventing the displacement of lines whose address are collected in the SR — for as long as remote disambiguation is in progress. This is the approach used in HTM and TLS systems. We do not support this approach.

5.6 Exceptions

When a conflict is detected on an SR, an exception may be triggered. The SPM supports registering exception handlers in a table, as shown in Figure 5 as *Exception Vectors*. If an exception is registered for a given SR, it is triggered when a conflict is detected. For a conflict caused by a local access, a precise exception is generated. For a conflict caused by an external coherence action, the handler is called as soon as the ICD or the SR detect the conflict.

When an exception is raised, the SPM notifies the processor's front end of the target address and informs the ROB of the instructions that need to be flushed. The exception handler then pushes the return address into a register and disables the handling of additional exceptions. Since other SRs may still be under disambiguation, additional exceptions are buffered and serviced sequentially.

6. MemoiSE: Signature-Enhanced Memoization

Memoization has been used to replace redundant or precomputed function calls with their outputs [17]. However, in languages such as C and C++, function memoization is hard to apply because memory state is often changed through non-analyzable pointer accesses. Using SoftSig, however, we propose a very general, low-overhead, and effective approach to increasing the number of function calls that can be memoized. We call our approach *MemoiSE*, for Signature-Enhanced memoization. In this section, we describe MemoiSE's general approach, the MemoiSE algorithm, and some optimizations to reduce its overhead.

6.1 A General Memoization Framework

Memoization algorithms work by caching the values of the inputs and outputs of a function in a lookup table. When the function is next invoked, the lookup table is searched for an entry with an identical set of input values. If such an entry is found, the output values are copied out of the lookup table into the appropriate locations (memory or registers), and the function execution is skipped.

Unfortunately, a function's inputs and outputs are not just the explicit input arguments passed to the function and the explicit output arguments returned by the function. They also include implicit inputs and outputs. These are other variables that the function reads from memory or writes to memory. To build a generic memoization algorithm, both explicit and implicit inputs and outputs need to be considered.

A naive approach would log the values of all implicit inputs and outputs in the table, in the same way as explicit inputs and outputs are logged. Unfortunately, implicit inputs and outputs cannot always be determined statically by the compiler, and there can be a very large number of them.

With MemoiSE, we do not log implicit inputs and outputs. Instead, we note that, if none of the implicit inputs or outputs have been written to since the end of the previous invocation of the function, then they have the same values. Such a condition can be easily checked using SoftSig. Indeed, during the initial execution of the function, we collect the addresses of all the implicit inputs and outputs in signatures. After the function completes, as the processor continues execution, the hardware enables the disambiguation of these signatures against all processor accesses. If, by the time execution reaches another call to the function, no conflicts have been discovered, it is safe to assume that the implicit inputs and outputs have not changed.

It is possible that, during the execution of the function, an implicit output overwrites a location read by an implicit input. In this case, since an input has changed, memoization should fail. We call this case *Internal Corruption*, and it must be detected to guarantee correctness of the optimization. Fortunately, detecting this case is easy with SoftSig signatures.

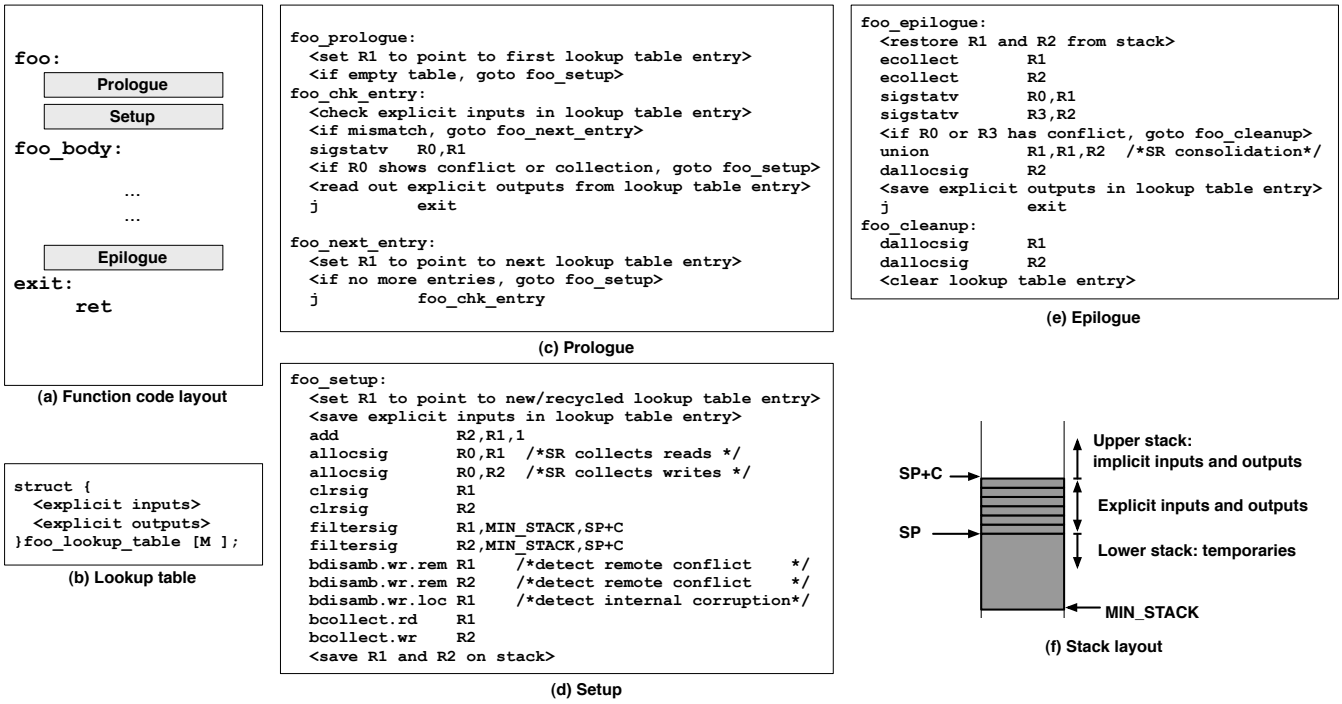


Figure 8. Applying the MemoiSE algorithm to function `foo`: function code layout (a), lookup table (b), Prologue (c), Setup (d), Epilogue (e), and stack layout (f).

In summary, MemoiSE works by recording the explicit inputs and outputs of a function call in the lookup table, collecting the addresses of the implicit inputs and outputs of the function using signatures and, after the function is executed, disambiguating these signatures against the addresses accessed by the code that follows. When we reach the next call to the function, we successfully memoize it if: (1) the explicit inputs match an entry in the lookup table, (2) during function execution, implicit outputs did not overwrite any implicit inputs, and (3) the implicit inputs and outputs have not been modified since the previous call as determined by signature disambiguation.

6.2 The MemoiSE Algorithm

MemoiSE is implemented by intercepting function calls using code inserted in functions. Figure 8 shows the application of MemoiSE to function `foo`. Part (a) shows the resulting layout of `foo`'s code. MemoiSE inserts three code fragments: Prologue, Setup, and Epilogue. Part (b) 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 the need to synchronize on access to a shared table. In the following, we explain the Prologue, Setup, and Epilogue code fragments. Note that we have skipped some code optimizations in the figure to make the code more readable.

6.2.1 Prologue

The Prologue is shown in Figure 8(c). It determines whether the call can be memoized and, if so, it 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 was used to collect the function's memory accesses when the function was called with the explicit inputs stored in the entry. Moreover, this SR has been disambiguated against all local and remote accesses since that function call was executed. Finally, the name of this SR was set to be the virtual address of the lookup table entry.

Based on this organization, the code in Figure 8(c) first sets register *R1* to point to the first entry in the lookup table, which is also the name of the associated SR. Then, the function's explicit inputs are compared to the values stored in the table entry. If they are the same, then this entry's explicit outputs can potentially be reused. However, we first need to check that the associated SR has not recorded a conflict since the function was last called with these explicit inputs. To perform the check, we first use the `sigstatv` instruction to read out the SR's Status Vector into register *R0*. If the bits in the Status Vector show both that there has been no conflict and that this SR is not currently collecting addresses (if it is still collecting, it would mean that the function is recursive and, therefore, cannot be memoized), then memoization succeeds. In this case, the explicit outputs are read out from the table entry and control transfers to the function return. If, instead, memoization fails, the function needs to be executed. Also, if the explicit inputs did not match, we check subsequent table entries until a match is found or the table is exhausted.

6.2.2 Setup

If the function call is not memoized, the Setup code fragment initializes the necessary structures to record the effects of this call. The code is shown in Figure 8(d). It involves three operations, namely 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 addresses of the implicit inputs and outputs.

The instructions for the third operation are shown in Figure 8(d). We allocate two SRs — one for addresses read and one for addresses written. In the figure, the name of the SR for reads is the address of the table entry, and it is stored in *R1*; the name of the SR

for writes is obtained by adding 1 to the entry's address, and 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. The stack locations where such variables are allocated is shown in Figure 8(f): we store explicit inputs or outputs between *SP* and *SP+C*, and temporaries between *MIN_STACK* and *SP*. In Figure 8(d), 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 (Section 6.1). Note that 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, since function `foo` may write these registers, and we will need them later.

6.2.3 Epilogue

After `foo` executes, we can fill the entry in the lookup table. This process is performed by the Epilogue as shown in Figure 8(e). This code first restores *R1* and *R2* from the stack and ends collection for both SRs. It then obtains the Status Vectors of the SRs and checks that they have not recorded a conflict. If either one has recorded a conflict, then memoization is not possible; we discard the entry in the lookup table and deallocate the two SRs.

Otherwise, the two SRs are consolidated into one SR (whose name is in *R1* in the example) to save space. Moreover, the explicit outputs of the call are saved in the entry of the lookup table. Note that the remaining SR is currently under disambiguation against local and remote writes.

6.3 Optimizations for Lower Overhead

Since only some functions can benefit from memoization, a profiler should identify which functions are amenable to memoization and apply MemoiSE only to them, as per **G4**. We leverage an analytical model proposed by Ding and Li [8] to identify which functions are most likely to benefit from memoization.

Furthermore, searching a large lookup table usually adds significant overhead. Consequently, we use profiling to discover which functions mostly need a single-entry table. For these functions, we restructure the table while providing space for only a single entry, so that the table can be accessed with very low overhead.

7. Evaluation

7.1 Experimental Setup

To estimate the potential of MemoiSE, we implemented an analysis tool that uses Pin [16], a software framework for dynamic binary instrumentation. The output of Pin is connected to a simulator of a multiprocessor memory subsystem based on SESC [23]. The simulator models per-processor private L1 caches attached to a shared L2 cache. Some parameters of the architecture are shown in Table 3. With this setup, we can estimate MemoiSE's reduction in number of instructions executed and in execution time. The latter is obtained assuming that, when memory accesses do not stall the processor, the average IPC of non-memory instructions is 1. We model the overlap of instructions with L2 misses.

For our experiments, we run the applications shown in Table 4. They are Firefox, Gaim, Impress, SESC and Supertux. The first three are popular applications used on many personal computers. SESC is an architectural simulator [23] and Supertux is an open-source arcade game. Of these applications, Firefox, Impress, and Supertux are multithreaded, and run with 6, 6, and 2 threads, respectively. For each application, we trace an execution of over 400 million instructions.

Reorder buffer	50 entries
Signature register file	16 signature registers, 1Kbit each
L1 cache: size, line, assoc, lat.	64 KB, 64B, 4, 1 cycle
L2 cache: size, line, assoc, lat.	2 MB, 64B, 8, 10 cycles
Max. outstanding L2 misses	16
Memory latency	500 cycles

Table 3. Parameters of the architecture simulated.

App. (Num Threads)	Description	Section Analyzed
Firefox (6)	Popular web browser	Begins after initialization, while it loads the *iacoma.cs.uiuc.edu* webpage
Gaim (1)	Open source instant messaging program	Begins once a client is running. It consists of opening a new message window, sending a message, and receiving a message
Impress (6)	OpenOffice presentation software	Begins with opening a sample presentation and continues while a user interacts with it
SESC (1)	Architectural simulator available from SourceForge.net	Performs a functional simulation of the *mcf* program using the default simulator configuration from SourceForge.net
Supertux (2)	"Jump'n run" arcade sidescroller game like Mario Brothers	Performed during game play. It begins when the penguin drops to the ground. It continues until the penguin dies and respawns

Table 4. Applications studied.

We study MemoiSE in the context of the four environments of Table 5. By default, the results are normalized to *Baseline*.

Environ.	Description
Baseline	No MemoiSE
Plain (P)	MemoiSE applied selectively to some functions using a cost-benefit analysis as in [8]. 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

Table 5. Environments analyzed.

7.2 Impact of MemoiSE

Figure 9 shows the dynamic instruction counts of the *P*, *O*, and *I* environments relative to *Baseline*. The figure shows data for each application and the average. Each bar is divided into two segments. The segment above zero (*Gains*) is the fraction of application instructions eliminated by memoizing function calls. The segment below zero (*Overhead*) is the additional instructions added by the memoization algorithm. The difference between *Gains* and *Overhead* is the savings achieved by MemoiSE, and is shown above each bar. A better optimization will have a taller *Gains* and a shorter *Overhead*.

From the figure, we see that, on average, *P* eliminates 13% of the application instructions. However, it adds overhead instructions, resulting in an average net instruction reduction of only 5.9%. For

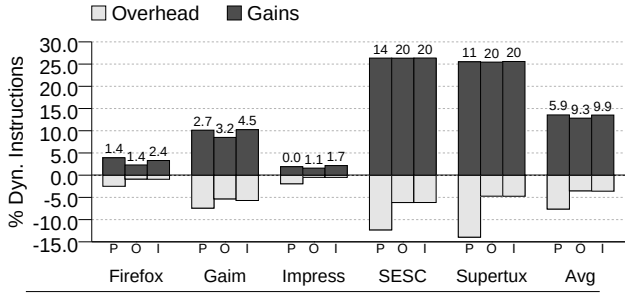


Figure 9. Dynamic instruction count relative to *Baseline*.

SESC and Supertux, the reduction in application instructions is especially significant, reaching over 25%. Moving now to the *O* bars, we see that keeping most of the tables to a single entry results in much lower overheads, while, in most cases, still eliminates a similar number of application instructions. The result is that the average net instruction reduction is lifted to 9.3% — and about 20% for SESC and Supertux.

Interestingly, having an unlimited number of SRs with no false positive conflicts (*I* bar) offers little additional advantage. The average net instruction reduction is slightly under 10%. Therefore, 16 SRs of 1Kbit each appear to be enough.

Figure 10 shows the execution times of the *P*, *O*, and *I* environments relative to *Baseline*. The figure shows that the execution time reductions closely follow the reductions in instruction count. On average, *O* offers a 9% reduction in execution time. Moreover, the reduction reaches 19% 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 *I* environment. They are slightly better than for the *P* environment.

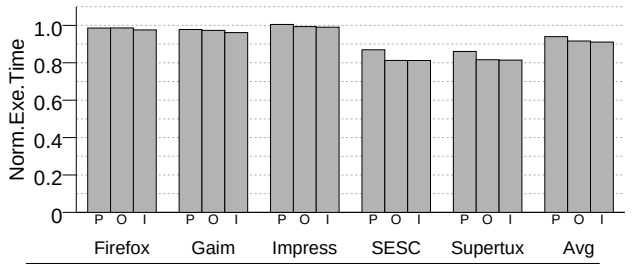


Figure 10. Execution time relative to *Baseline*.

Our execution time analysis provides insights into the overheads of MemoiSE. Because MemoiSE adds a lookup table for each memoized function, the overheads of memoization can vary depending on the cache behavior of the accesses to the lookup tables. Fortunately, we observed that the accesses to the lookup tables rarely caused L2 misses due to the temporal locality of function calls.

Figure 11 shows the contention on the SRF. For each application and for the average, the figure shows the average number of accesses per cycle to the SRF for collection and disambiguation. Each bar is broken into four categories of accesses: *C-Rd* is the collection of read addresses, *C-Wr* is the collection of written addresses, *D-Loc* is local disambiguation, and *D-Rem* is remote disambiguation. There are also additional accesses due to allocating, deallocating, and manipulating SRs. However, they are not seen in the figure because they account for a very small fraction of the total accesses.

From the figure, we see that the average number of SRF accesses per cycle is about 0.11. This means that the SRF is only accessed roughly once in 10 cycles. This is a tolerable access frequency. In addition, for the multithreaded applications (Firefox, Impress, and Supertux), remote disambiguation causes most of the accesses. In all the applications, collection and local disambiguation are less significant, in part due to the filtering of many stack accesses.

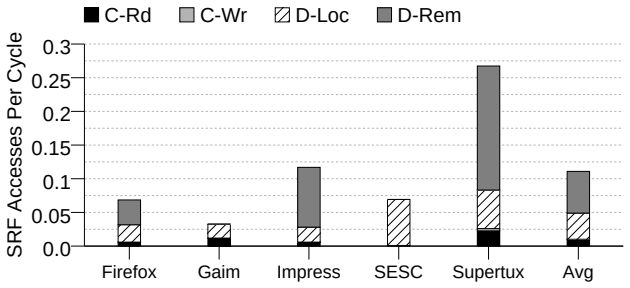


Figure 11. Mean number of SRF accesses per cycle of execution.

7.3 Function Characterization

To further understand MemoiSE, we analyze in detail one function that is frequently memoized from each application. The functions are shown in Table 6. From left to right, the columns of the table show: (1) function name; (2) application name; (3) explicit inputs; (4) type of the explicit output; (5) number of calls to the function in the execution analyzed by Pin; (6) average dynamic size of each call in instructions; (7) fraction of the total application instructions eliminated by memoization (*Gains* in Figure 9) that are contributed by this function; (8) fraction of the calls to the function that are memoized; (9) fraction of the failed memoizations of this function due to conflicts that are caused by false positives; and (10) average read and (11) write set size, respectively, of the function when it is successfully memoized. The read (or write) set size is the number of reads (or writes) to different locations.

`g_value_type_compatible` from Firefox is a function in the GLib GTK+ core library that checks whether two object types are compatible with each other. The check is done by accessing a type table and examining the inheritance tree. As shown in the *Mem* column, memoization is typically successful (75% of the times). This is because the data structures are only updated when a type is first registered. However, the large variation in the input values sometimes causes misses in the lookup table.

`pango_fc_font_get_glyph` from Gaim is a Pango GTK+ font library function that gets the glyph index of a given Unicode character for a font. The properties of a glyph within a font do not change once the font is loaded into memory and are requested frequently in Gaim as each character is processed. Memoization is often successful (21% of the times, as shown in the *Mem* column), as characters are repeated. However, a larger lookup table would be desirable for a longer history of characters.

`_dl_name_match_p` from Impress is a function used internally in the GNU C library to test whether the given name matches any of the names of the given object. It is often used to resolve the name of a dynamically-loaded object such as a shared library object. Since the list of names for an object is updated only when it is first registered (e.g., when a shared library is first loaded), this function behaves much like a pure function at runtime. Consequently, as shown in the *Mem* column, it is memoized 95% of the times.

Function Name	App.	Explicit Inputs	Explicit Output	#Calls (Thous.)	Size (Inst.)	Weight (%)	Mem (%)	FP (%)	R Set	W Set
g_value_type_compatible	Firefox	GType src_type, GType dest_type	gboolean	17.8	212	19	75	0	7	0
pango_fc_font_get_glyph	Gaim	PangoFcFont* font, gunichar wc	guint	25.1	322	5	21	1	29	0
_dl_name_match_p	Impress	const char* __name, struct link_map* __map	int	41.1	80	31	95	0	8	0
OSSim::enoughMTMarks1	SESC	this, int pid, bool justMe	bool	33481.0	35	80	100	0	2	0
Sector::collision_static	Supertux	this, collision::Constraints* constraints, const Vector& movement, const Rect& dest, GameObject& object	void	7.7	5023	14	29	10	551	0

Table 6. Five functions that are frequently memoized from the different applications.

Moreover, as shown in the *Weight* column, it contributes 31% of the application instruction reduction.

`OSSim::enoughMTMarks1` from SESC monitors several conditions to determine when the program should begin and end detailed timing simulation. While the condition checks are optimized, they are performed frequently — some kind of check is required after each instruction is simulated. The function is only 35 instructions, but it is called millions of times. Its memoization is practically 100% successful, and it accounts for 80% of the application instruction reduction.

`Section::collision_static` from Supertux is part of the game logic that detects when collisions occur. It is called from three sites in the same function. The first two sites are in loops, with each iteration changing one of the input parameters. As a result, calls from these sites are not memoized. However, between the second and third sites, no change typically occurs to the parameters, allowing memoization. For these reasons, memoization is only 29% successful. Nonetheless, the function contributes 14% of the application instruction reduction because it has a large size (5023 instructions). However, a read set size of 551 addresses, as shown in the *R Set* column, leads to an increase in false positives. Indeed, Column *FP* shows that, of all failed memoizations due to conflicts, 10% are caused by false positives.

One additional observation is that all these functions have a zero write set when they are successfully memoized. We find that written locations are typically read by the same function, causing internal corruption and memoization failure.

8. Related Work

8.1 Signatures & Bloom Filters

SoftSig builds on the body of work that uses hardware signatures and Bloom filters for efficient disambiguation (e.g., [3, 4, 18, 19, 21, 25, 31]). Bulk [3], LogTM-SE [31], and SigTM [18] are closely related to SoftSig. Each system uses signatures for the explicit purpose of supporting TM or TLS. For the case of Bulk and LogTM-SE, these signatures are hardware registers that are used for the sole purpose of logging memory accesses and performing conflict detection. LogTM-SE can save and restore signatures to support virtualization, but in no other way are they manipulated by software.

SigTM employs a limited set of signature operations in software to implement Software Transactional Memory. Software can insert an address into a signature or do a membership test to support read and write barriers. Software can also do remote disambiguation to detect conflicts between transactions. SigTM, like LogTM-SE, has the means to save or restore a signature.

However, none of these schemes provide a comprehensive ISA to manipulate multiple signatures in a register file that enables the wide variety of tasks discussed in Section 3. Furthermore, SoftSig can be used even without support for speculative execution as in MemoiSE. Per **G1**, we opted not to save and restore SRs for the scenarios we considered. However, SoftSig does support save and restore, and can use them if SoftSig were employed in a TM system.

8.2 Memoization

Memoization has been studied at the granularity of instructions [14, 15, 26, 27] and coarse-grained regions [5, 6, 10, 24, 30]. Sodani *et al.* [27] empirically characterized the sources of instruction-level repetition and some characteristics of function-level behavior. They found that a large number of dynamic function calls are called with repeated arguments, and that most of these calls had either implicit inputs or side effects. This lead them to conclude that few functions could be memoized. However, with SoftSig, implicit inputs and side effects are easily coped with.

Connors *et al.* [5, 6] studied memoization of coarse-grained regions of code using a compiler (augmented with profiling information) to identify profitable regions. During execution, compiler-inserted instructions direct the hardware to record the explicit inputs and outputs for a region in a hardware table. Then, when the region is encountered again, the table is checked for a solution. If one exists, the outputs are written into registers directly by the hardware, and the region is skipped. To account for memory inputs, each table entry has a memory valid bit which is cleared anytime a memory input for that entry is potentially updated. The compiler is responsible for scheduling invalidate instructions in the code. Wu *et al.* [30] built on top of [5] by combining speculation and memoization to exploit more region-level reuse.

MemoiSE differs from all of these in that it only targets functions, as opposed to arbitrary regions of code. MemoiSE does have an advantage, in particular over [6], in that it can dynamically detect any memory access that invalidates an entry in the lookup table. In addition, the memory accesses of a function do not need to be analyzed statically for correct memoization. MemoiSE incurs overhead for table lookups in software. Such lookups are done in hardware in [6]. If MemoiSE lookups were done in hardware, the overheads could be significantly reduced.

Ding and Li [8] proposed a compiler-directed memoization scheme implemented fully in software. The compiler identifies coarse-grained regions of code for reuse and then generates the necessary code to store the inputs in a lookup table and check the table on future calls. The compiler must prove that all inputs are invariant for a memoized region. Also, because there is no hardware support, the compiler must perform a cost-benefit analysis to decide when a region of code is worth memoizing. MemoiSE is similar to this approach in that the lookup table is a software structure and the compiler/profiler must decide which functions to transform using a similar cost analysis. MemoiSE, however, can more aggressively select functions since implicit inputs and outputs are checked dynamically.

9. Conclusion

This paper proposed the SoftSig architecture to enable flexible use of hardware signatures in software for advanced code analysis, optimization, and debugging. SoftSig exposes a Signature Register File to the software through a rich ISA. The software has great flexibility to decide: (i) what stream of memory accesses to collect in each signature, (ii) what local or remote streams of memory accesses to disambiguate against each signature, and (iii) how to manipulate each signature. We also described the processor extensions needed for SoftSig.

In addition, this paper proposed to use SoftSig to detect redundant function calls efficiently and eliminate them dynamically. We called our memoization algorithm MemoiSE. Our results showed that, for five multithreaded and sequential applications, MemoiSE reduced the number of dynamic instructions by 9.3% on average, thereby reducing the average execution time of the applications by 9%.

SoftSig can be used for many other optimizations. Several proposals for runtime-disambiguation based optimizations can be revisited, with potentially new applications or more general use [1, 13, 22, 29]. Also, aggressive speculative optimizations based on checkpointing [20] may benefit from SoftSig's ability to record information about a program's dependences. Of course, SoftSig can integrate into environments that already use signatures [3, 18, 31] to enhance the software's or the programmer's control over signature building and disambiguation.

References

[1] D. Bernstein, D. Cohen, and D. E. Maydan, "Dynamic Memory Disambiguation for Array References," in *International Symposium on Microarchitecture*, November 1994.

[2] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 11, July 1970.

[3] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *International Symposium on Computer Architecture*, June 2006.

[4] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *International Symposium on Computer Architecture*, June 2007.

[5] D. A. Connors, H. C. Hunter, B.-C. Cheng, and W.-M. W. Hwu, "Hardware Support for Dynamic Activation of Compiler-Directed Computation Reuse," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[6] D. A. Connors and W.-M. W. Hwu, "Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results," in *International Symposium on Microarchitecture*, November 1999.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2001.

[8] Y. Ding and Z. Li, "A Compiler Scheme for Reusing Intermediate Computation Results," in *International Symposium on Code Generation and Optimization*, March 2004.

[9] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-M. W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[10] J. Huang and D. Lilja, "Exploiting Basic Block Value Locality with Block Reuse," in *International Symposium on High Performance Computer Architecture*, January 1999.

[11] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part II*, November 2007.

[12] V. Krishnan and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Trans. on Computers*, September 1999.

[13] J. Lin, T. Chen, W.-C. Hsu, and P.-C. Yew, "Speculative Register Promotion Using Advanced Load Address Table (ALAT)," in *International Symposium on Code Generation and Optimization*, March 2003.

[14] M. Lipasti, C. Wilkerson, and J. Shen, "Value Locality and Load Value Prediction," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[15] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit Via Value Prediction," in *International Symposium on Microarchitecture*, December 1996.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *International Conference on Programming Language Design and Implementation*, June 2005.

[17] D. Michie, ""Memo" Functions and Machine Learning," in *Nature*, April 1968.

[18] C. C. Minh *et al.*, "An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees," in *International Symposium on Computer Architecture*, June 2007.

[19] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi, "JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers," in *International Symposium on High-Performance Computer Architecture*, January 2001.

[20] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles, "Hardware Atomicity for Reliable Software Speculation," in *International Symposium on Computer Architecture*, June 2007.

[21] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai, "Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching," in *International Conference on Supercomputing*, June 2002.

[22] M. Postiff, D. Greene, and T. Mudge, "The Store-load Address Table and Speculative Register Promotion," in *International Symposium on Microarchitecture*, December 2000.

[23] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC Simulator," January 2005. http://sesc.sourceforge.net.

[24] S. Sastry, R. Bodik, and J. Smith, "Characterizing Coarse-Grained Reuse of Computation," in *Workshop on Feedback-Directed and Dynamic Optmization*, 2000.

[25] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler, "Scalable Hardware Memory Disambiguation for High ILP Processors," in *International Symposium on Microarchitecture*, December 2003.

[26] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *International Symposium on Computer Architecture*, June 1997.

[27] A. Sodani and G. S. Sohi, "An Empirical Analysis of Instruction Repetition," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[28] G. Sohi, S. Breach, and T. Vijayakumar, "Multiscalar Processors," in *International Symposium on Computer Architecture*, June 1995.

[29] B. Su, S. Habib, W. Zhao, J. Wang, and Y. Wu, "A Study of Pointer Aliasing for Software Pipelining Using Run-time Disambiguation," in *International Symposium on Microarchitecture*, November 1994.

[30] Y. Wu, D.-Y. Chen, and J. Fang, "Better Exploration of Region-level Value Locality with Integrated Computation Reuse and Value Prediction," in *International Symposium on Computer Architecture*, June 2001.

[31] L. Yen *et al.*, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *International Symposium on High Performance Computer Architecture*, February 2007.