

Hardware Support for Pointer Safety in Commodity Microprocessors

Diana Keen, Foo Lim, Frederic T. Chong, Premkumar Devanbu, Matthew Farrens,
Paul Sultana, Cathy Zhuang, and Ravishankar Rao
Computer Architecture Laboratory and Software Tools Laboratory
University of California at Davis

Abstract

Rising chip densities have led to dramatic improvements in the cost-performance ratio of processors. At the same time, software costs are burgeoning. Large software systems are expensive to develop and are riddled with errors. Certain types of defects (e.g., those related to memory access, concurrency, and security) are particularly difficult to locate and can have devastating consequences. We believe it is time to explore using some of the increasing silicon real-estate to provide extra functionality to support software development. We propose dedicating a portion of these new transistors to provide hardware structures to enhance software development, make debugging more efficient, increase reliability and provide run-time security.

Unfortunately, software safety tasks, such as checking pointer references, involve extensive book-keeping which result in unacceptably high overheads. We introduce a *hardware access table* (HAT) which accelerates table lookups critical to software monitoring tasks. We propose two architectures, one general-purpose and one special-purpose, that utilize the HAT and analyze their effectiveness. The special-purpose HAT sustains good performance as load increases, while the general-purpose HAT performance degrades quickly. In particular, the special-purpose HAT increases performance by up to 290% over software-only pointer checking with only a 1.7% investment in processor area and minimal increase in processor design complexity.

1 Introduction

For several decades, advances in chip manufacturing technology have provided designers with an ever-increasing pool of available transistors. Designers have tended to use these transistors primarily to provide performance gains, yielding such innovations as out-of-order execution, multiple-issue pipelines, on-chip multi-threading, larger caches, branch prediction logic, etc.

At the same time, software development costs have skyrocketed. Large, complex software systems such as operating systems, databases, switching systems

and desktop productivity applications are expensive, often behind schedule, and plagued with defects. In fact, maintenance and defect-removal costs are the major factors in the cost of developing large software systems. When these contrasting trends in hardware and software are juxtaposed, the question naturally arises: can some of the increasing bounty of real-estate available on current and future chips be harvested to provide support for software development activities? Specifically, we propose to increase the productivity of software engineers by providing pointer safety with acceptable performance. Our work also makes progress towards providing secure systems at run time.

The specific software defects we will detect are dynamic memory access defects. These arise when a piece of data intended to be of one particular type is used as a different, incompatible type. This type of error has been called “unsafe” by Cardelli [1], and the effect is not always immediate and noticeable. The resulting failure may not occur until after quite a bit of additional computing has occurred, and the symptoms may have become much more diffuse. As a result, these defects are particularly pernicious and difficult to track down. Pointer defects are also commonly exploited for security violations, a scenario exemplified by the finger daemon benchmark used in this study. Our pointer safety system can detect exactly these vulnerabilities either at development or run time.

Languages like Java combine static and dynamic support to prevent these errors. Unfortunately, because of legacy assets, efficiency, and available talent, it is likely that systems written in type-unsafe languages like C or C++ will remain under active development or maintenance for the foreseeable future. In C and C++, the only way to detect memory access errors immediately is at run-time, verifying the validity of a pointer value before it is dereferenced. This dynamic checking has a significant performance impact.

The performance impact of dynamic checking in C and C++ code can be reduced through careful addition of hardware resources to commodity microprocessors. A critical issue is reducing the overhead of book-keeping operations. Towards this end, we introduce a *hardware access table* (HAT) which efficiently

implements an associative memory by leveraging the existing level 2 cache. We demonstrate that this simple mechanism substantially reduces the overhead of our dynamic checking tasks, and expect in the future other types of applications may be able to exploit this structure in the future as well. Two versions of the HAT are presented here, one with a generalized instruction interface making it accessible to user programs, and one with a memory-mapped I/O interface to a specialized HAT usable only for background monitoring activities such as dynamic pointer checking. In programs with substantial overhead, the specialized HAT reduces the runtime by a factor of 2 to nearly 3.

We begin in Section 2 by presenting several previous approaches to memory safety. Section 3 describes the software baseline we chose for our system. We present our hardware access table (HAT) in Section 4, followed by a description of our two test architectures that utilize the HAT in Section 5. Our applications are described in Section 6. We look at the performance data in Section 7 and discuss the implications and future directions in Section 8. Section 9 gives our conclusions.

2 Background

The focus of this study is hardware support for dynamic pointer checking on a commodity microprocessor. Our work builds upon static analysis approaches to inserting dynamic checks in software by investing hardware to accelerate these checks. Our problem can also be solved by specialized architectures, such as tagged processors, but our explicit goal is to support commodity microprocessors.

Static analysis of source [2] has been used to find potential errors such as buffer overflows. This approach (like other approaches based on static analysis) is plagued by undecidable sub-problems, and thus is forced to make worst-case assumptions. This unfortunately can result in numerous false positives. Although such approaches promise more focused manual inspections, numerous false-positives will lead to wasted effort; busy programmers may balk at using such a tool.

A more practical approach is to use static analysis to generate dynamic checks which can detect errors such as array-bounds overflow or security policy violations at run-time. Approaches include source- or binary-instrumentation [3], or run-time environments such as the Java Virtual Machine. For type-unsafe languages, *augmented pointers* provide a mechanism to support dynamic checking of safety. In this ap-

proach [4, 5, 6], pointers record bounds information for an object. Our work starts with [4], a leading approach for single-processor systems, and focuses on how to improve the performance by using our hardware structures.

Specialized architectures, such as Sun's Pico-Java for Java, Intel's iAPX432 [7], and the Symbolics LISP machines [8], provide special support for tagging and checking in hardware. Changing a commodity processor such as the Intel Pentium, however, to include such mechanisms would be prohibitively expensive. Our goal is to achieve acceptable performance with minimal modification to a commodity processor.

3 Baseline Software Implementation

Normally, a pointer is associated with a memory location of a particular type. If a pointer should (through accident or malice) point to a location of a different type, it may be said to be *unsafe*. Unsafe pointers are a common source of mysterious software failures. Memory corruption due to an unsafe pointer is notoriously difficult to trace because the effects may not become apparent for an arbitrarily long time after the first pointer wrote in the wrong location. In addition, programmers often fail to check user input before use. Particularly with arrays, this can leave programs vulnerable to attack resulting from buffer overflows. Both of these types of mistakes may require specific input conditions to trigger, and are easily missed in testing.

For our study, we begin with the augmented pointer representation as proposed in [4]. Each pointer has associated with it information about the object to which it points. This *Safe Pointer* structure contains the current value of the pointer, the base (starting address) of the object, the size of the object, and a unique identifier (ID) of the object.

Each time a pointer is used, two checks must occur. A spatial validity check is performed to verify that the value of the pointer is within the bounds of the object. A temporal validity check is also performed to confirm that the referred object is still live, *i.e.*, it has not been freed and perhaps re-allocated for a new object. The information required for the spatial check is contained within the augmented pointer data structure. This information is insufficient, however, to determine temporal validity. A single object might have several pointers referring to it, and a deallocation can not update all of the pointer structs. For temporal validity, we must maintain a separate *temporal hash table* that indicates whether an object with

Event	Monitoring Action	ID Table Action
Spatial	Spatial check	
Liveness		Find ID
Alloc	Fill in struct	Insert ID
Dealloc	-	Remove ID
Assign	Copy struct	-
Call	Push ID	Insert ID
Return	Pop ID	Remove ID
setjmp	mark stack	
longjmp	Pop IDs	Remove ID(s)

Table 1: Checking Actions

a specific ID is still allocated and has not yet been freed. To correctly perform the checks each time a pointer is used, these data structures must be kept up to date. Whenever a pointer is assigned to another pointer, the safe pointer structure is copied automatically. When a pointer is assigned to the address of an object, the information about the object must be placed in the safe pointer structure.

The unique IDs are generated by a global counter, which is incremented upon every object allocation. The counter is also incremented when a new stack frame is created in response to a procedure call. Local, stack-allocated objects are given the ID of the current active stack frame. Stack object sizes are calculated at compile-time unless the allocation routine is called, in which case the routine is instrumented. The current stack ID and base must be obtained at run-time. Heap allocated object information is also determined at run-time. All of the information for global objects is known at compile-time. As long as an object is active, its ID is resident in the temporal hash table; once the object is freed (upon explicit deallocation for heap objects and a stack pop for stack objects) the ID is removed from the hash table.

The code must be instrumented to both maintain these data structures and check pointer validity. Static analysis is performed on the source code to identify (conservatively) the relevant locations, and we then insert instrumentation as summarized in Table 1. This purely software-based mechanism is then accelerated by specialized hardware components that we introduce in Section 4.

4 Hardware Access Table

Our work begins with an analysis of the factors contributing to the overhead of pointer safety. We break the execution down into four categories: Base, Struct,

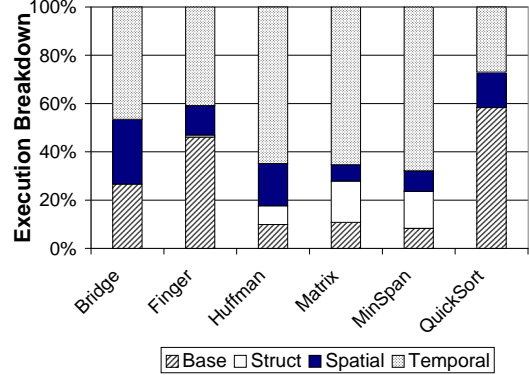


Figure 1: Execution Breakdown of Instrumented Applications. Base is the uninstrumented application, Struct is the additional overhead due to increasing the size of each pointer, Spatial is the time for spatial checks, and Temporal is the time due to temporal checks and the bookkeeping associated with them.

Spatial, and Temporal. Base is the original, uninstrumented application. Struct is the overhead from increasing the size of each pointer. Spatial is the time contributed by spatial checks, and Temporal includes both the temporal checks and the bookkeeping associated with them. We see in Figure 1 that the temporal hash table operations (allocations, deallocations, and liveness checks) comprise the largest fraction of the pointer safety overhead.

A hash table access has two parts - the original query into the location mapped by the hash function and a conflict resolution protocol when multiple items hash to the same location. In order to reduce the impact of these hash table operations, we propose using a hardware structure called a Hardware Accelerated Table (HAT). The HAT has a “bucket” for each hash entry that contains five items - in our studies we chose to store the four most recently used IDs along with a link to conventional memory (stored in the L2 cache) that holds the next “bucket” to search.

The HAT can be thought of as a cache for hash table lookups coupled with the logic to find elements that have been evicted from this cache. All references to the temporal hash table will access the HAT first, and then access the HAT overflow area in the L2 cache if the pointer is not found in the HAT. Unlike ordinary caches, however, the HAT also provides hardware acceleration for standard hash table functions - Insert, Remove, and Find. When an item is not found in the HAT, the corresponding overflow

line in the L2 cache is brought into the HAT, and the acceleration hardware is used to search the new line quickly and efficiently.

Figure 2 presents an example of how the HAT works. In the figure, the four items in bucket three are read out and compared in parallel to ID 0x5503 in a manner entirely analogous to how a standard 4-way set associative cache operates. If none of the four comparisons succeed, then the next bucket is brought in from the L2 cache into set “B”, and the comparators are used to again perform a parallel search of the bucket. This process repeats until either the desired pointer is found, or the end of the linked list is encountered.

Several issues influenced the design of the HAT:

- The physical size of the HAT is static
- The access time of the HAT is proportional to its size
- Placing an extra component in the core of the processor will most likely increase wire delays, slowing down the processor clock and substantially increasing processor complexity and design time.
- Placing an extra component away from the core of the processor will increase the delay for accessing that particular component.

In order to reduce the effect on the processor design, we placed the HAT next to the L2 cache. This makes the delay to and from the processor large but reduces the delay for fetches into the cache. Our results will show that the processor to HAT latency does not have a significant affect on performance.

The hash table is mapped onto hardware in the following fashion: The hash function is a mask of bits 0 through x-1, where there are 2^x sets in the HAT cache. This hash function determines in which set in the HAT to search. The search proceeds like a normal cache access - parallel hardware is employed depending on the degree of associativity in the cache. Our simulations showed that there was very little increase in hit rate when moving from four to eight-way associativity, so we chose four-way associativity.

4.1 Hat Miss Resolution

The HAT acts like a cache in terms of searches within the HAT and deciding what elements to evict. A difficulty arises, however, when we try to evict an element from the cache or we experience a miss on a search. With conventional caches, there is no problem. The element is sent to the lower level of memory, and it is

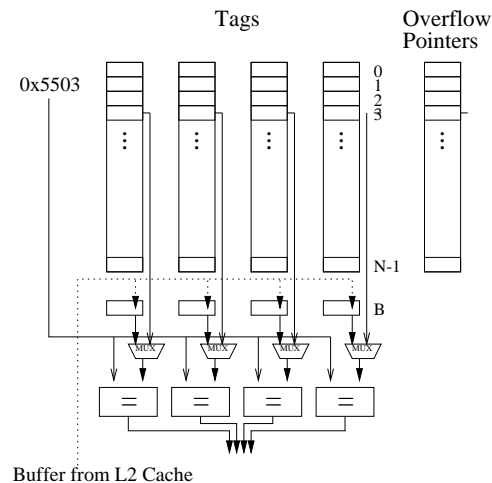


Figure 2: The HAT Architecture

placed in a location dependent on its key (the memory address). In our case, the key is unrelated to a memory address, so we can not evict our element to the L2 cache using only the key.

The HAT must allocate overflow memory to store evicted entries. Each set has a pointer associated with it storing the memory address of its overflow memory. The memory for the HAT is allocated at startup for reasons addressed later. The HAT manages its own memory internally. A single line is allocated and initialized each time a set runs out of empty slots in its current set of overflow buckets. Subsequent HAT cache misses must search the overflow to find the relevant entry. If the item is not in the HAT's cache, we read in the overflow line from the L2 cache. We continue to use our 4-way associative comparison for overflow entries. Figure 2 shows the overflow pointers as well as temporary buffer in which lines from the L2 cache are held. When an L2 cache line is read in, four tags are compared simultaneously each cycle. The last entry in the cache line is a pointer that points to the location of the next set of overflow entries. The search is terminated when the element is found or the spillover region has been exhausted. If an L2 cache line holds more than four entries, multiple sets of temporary buffer space are provided. Thus, on a miss, the HAT has two jobs: Search the current overflow line to find a match and request the next overflow line from the L2 cache.

For reasons explained earlier, the HAT resides next to and interfaces with the L2 cache. This can introduce consistency issues if two things occur: The L1 cache is not write-through *and* the user code itself

is altering the HAT internal overflow area. No valid user pointer may point to HAT internal data, so our safepointer system would discover and flag this erroneous access before it occurred.

4.2 Hat Virtual Memory

Because the L2 cache is physically addressed, the HAT must use physical addresses in requests for memory overflow lines. There are two choices: Provide a TLB or pin pages in memory. Our largest application used 212KB of memory, substantially smaller than the 4MB superpages that is supported by Pentium systems. In our design, we continue using physical addresses and require that the operating system pin this 4MB page in DRAM.

When the HAT runs out of memory, the designer may choose between two alternatives. The program may continue running with inaccuracy, possibly causing false positives (because an ID that was inserted into the table was dropped), or may restart the program with a larger initial allocation for the HAT.

If dynamic memory is deemed necessary, a TLB can be added to the HAT. Pages would be allocated at the superpages granularity in order to minimize the number of entries in the TLB. A page miss would be handled by the main processor, giving the processor direct access to TLB entries through memory-mapped I/O.

4.3 Security Model

Given our HAT assumptions, what components must we trust to be reliable or secure? As with the original software scheme [4], the compiler must be trusted to instrument the source code correctly. Because we will decouple our pointer checking functions from the main processor core, latency of checks becomes an issue. In order to tolerate this latency, checks will not complete at each pointer reference. Rather, checks will be initiated before the pointer dereference, but detection of errors will be delayed until after the pointer reference. Our system must guarantee that the erroneous pointer reference can not corrupt the data used in checking the access. Protecting monitor data and tolerating detection latency are the primary differences between a loosely-coupled (inexpensive) and a tightly-coupled approach. This detection latency, however, can be tolerated in most error detection tasks and designed into security tasks. We return to this issue in Section 5.2.

5 System Architectures

We investigated two systems utilizing the HAT: A generic structure, *GenHAT*, controlled with special instructions, and a specialized structure, *SpecHAT*, controlled by a monitor through memory-mapped I/O that may be used only for monitoring activities. We also examined the hardware structures' physical requirements.

5.1 GenHAT

The generic HAT will provide finds, insertions, and removals through a special instruction using a register-based interface:

```
HAT_find   <dest_register> <key_register>
HAT_insert <data_register> <key_register>
```

Notice that there is no HAT_remove instruction. When the HAT does not find an item, it returns NULL. Thus a HAT_remove has the same effect as HAT_insert NULL <key>.

The hash table instructions are replaced in the serial code with these special HAT instructions. Static analysis is utilized to reduce the number of spatial and liveness checks that must be performed. The system is shown in Figure 3 (The components are not shown to size). The HAT is only 4.5 K (plus the logic to allocate memory and make requests of L2 cache), so it is considerably smaller than the L1 caches shown. The code must wait for responses, but instructions may be reordered around the instructions to mask the delay. Because the HAT is located near the L2 cache, we allocate two cycles for requests to travel to the HAT and two cycles for results to return to the core.

In order to support speculation, we use the same approach as memory accesses, allowing us to share the speculation hardware with memory accesses. An insert is analogous to a store, and a find is analogous to a load. If an insert precedes a find, and both operations have the same key, then the find must return the value from the the insert. This works with removes, as well. Remove is an insert of the value NULL. The later find will return the inserted NULL, indicating a failure to find the item. The rules for matching finds and inserts are identical to those of loads and stores, except that instead of matching on the address, we are matching on the key.

In order to share the speculation support hardware, the memory hardware must obtain the key in the same position as the address is in loads and stores. The key is read in from a register, not calculated from

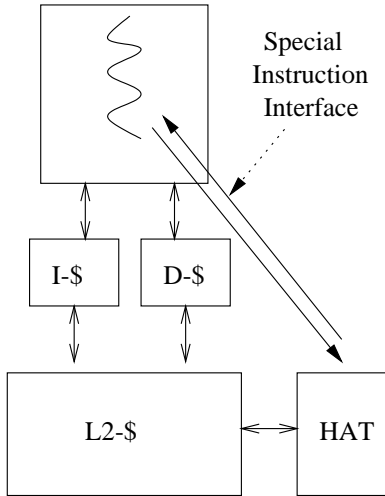


Figure 3: Single thread accesses Generic HAT through special instructions

a register and offset. Thus, the offset is always 0 in order to utilize the same instruction format. HAT and memory operations share the LD/ST reorder buffer. One bit in the instruction opcode as well as in the reorder buffer is added to distinguish between HAT and memory instructions. Since we are reducing overall memory traffic, we expect performance to increase despite sharing LD/ST resources. Each HAT command would have produced at least one (and often many) memory requests.

The generic HAT is similar in spirit to victim caches [9] and has a similar goal as pointer-based prefetching [10]. The HAT, however, takes an additional step in supporting comparison operations that provide the user an efficient implementation of an associative memory. With such a low cost and efficient mechanism, it is our hope that users will be free to choose the more natural model of keys and data over hashing and addresses, even in applications other than pointer safety.

5.2 SpecHAT

We can take a more aggressive approach by creating a specialized HAT controller. Instead of using new instructions for find, insert, and remove and then checking the result to make sure the operations are legitimate, we will offload this computation to a specialized checking engine, as shown in Figure 4. Requests are placed into a hardware queue using memory-mapped I/O. Each requests consists of two inserts into the queue. The first element is

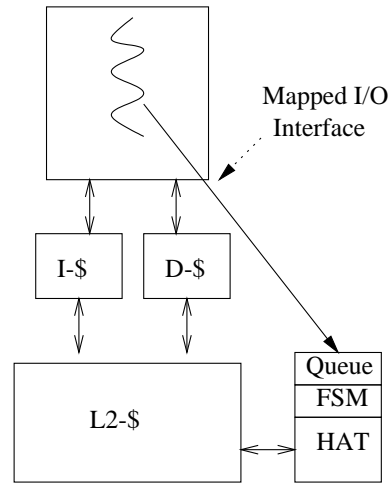


Figure 4: Single thread inserts into a memory-mapped hardware queue without waiting for a response. FSM receives input from hardware queue, makes requests of the Specialized HAT, and receives responses from the Specialized HAT.

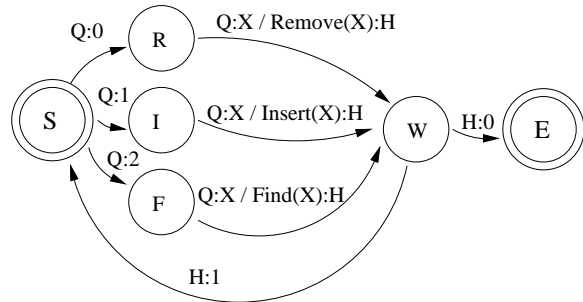


Figure 5: Finite State Machine that controls the HAT. The FSM receives input from both the Queue (Q) and the HAT (H). Q:X and H:X indicate an input of X from the hardware queue or HAT, respectively. X:H indicates a request of action X to the HAT. At any particular state, the FSM waits for input from either the HAT or the queue. If the FSM is not waiting for input from the queue, and there is data in the queue, the data will remain in the queue until the FSM is ready for it.

the type of request (find or insert), and the second is the tag (in our case the ID of the object). A simple state machine receives requests from the queue and controls the HAT. We call this hardware a *monitor*. The state machine is shown in Figure 5.

Using memory-mapped I/O simplifies the design because speculation is no longer an issue. Memory-mapped I/O operations are executed at commit time, meaning no speculative temporal check requests will be placed in the queue.

Allowing dereferences to commit before the temporal check occurs opens the possibility for a stray pointer to corrupt the overflow memory used by the HAT. For this to occur, however, the spatial check must be successful, because it is still performed before the instruction is allowed to commit. The only way for the spatial check to succeed on memory in the overflow memory is for the overflow memory to be using memory previous allocated to the program. Recall that HAT memory is allocated when the program begins, so no valid pointer can point to HAT internal data.

Limiting use of the specialized HAT to dynamic pointer checking allows several implementation optimizations. The advantages over GenHAT are reduced design and running time. Design time is reduced because changes only occur from the memory-mapped I/O interface out. This interface is outside the processor core, requiring no added instructions and hardware in the LD/ST queue. Run-time is reduced for four reasons. First and foremost, the SpecHAT reduces memory requirements. Pointer-checking requires the existence of a tag, not the data associated with a tag. This reduces the size of each element to one word, not two. Not only does this improve performance by packing twice as many elements in each L2 cache line, but it also reduces the total memory requirements for the HAT, reducing the amount of memory that needs to be pinned in DRAM. Second, the main process no longer needs to check what values come out of the HAT, so later instructions need not wait for the HAT to complete. Third, the state machine can transition in a single cycle, has no branch penalty, and has no instructions to produce instruction cache misses. Finally, we can optimize the delete instruction. Because an ID is only written upon insertion, the only time an element can be dirty is if it has not been evicted - thus, it has no entry in the overflow space. This means that if a delete occurs for a dirty element, we can skip the search in the overflow space.

App	Spatial	Live	Alloc	Delete
Bridge	2.4e5	8.2e4	1.3e4	1.3e4
Fingerd	8.4e3	5.9e3	101	100
Huffman	2.7e6	2.2e6	7.6e4	7.6e4
MatMul	5.1e6	7.3e5	5.4e4	2.7e4
MinSpan	2.2e5	1.8e5	8.6e3	8.6e3
Quicksort	2.9e5	1.0e4	3.0e4	3.0e4

Table 2: Dynamic characteristics of test programs

5.3 Physical Requirements

The specialized HAT was synthesized from VHDL to provide an upper bound on physical requirements. The request queue plus specialized HAT requires 0.7 million transistors. This is 1.7% of the number of transistors on the Pentium 4. An optimized, custom implementation would clearly require significantly fewer resources.

GenHAT is slightly larger with the removal of the request queue and state machine and the addition of twice the storage in the HAT cache. Furthermore, GenHAT requires changes to the core. An extra bit must be added to the LD/ST queue to distinguish LD/ST instructions from HAT instructions, the HAT must be connected to the LD/ST buffer, and decode logic must be added to recognize our two new opcodes.

6 Applications

To evaluate our monitoring mechanisms, we used several applications that require dynamic run-time checks to guarantee security. Library calls such as `gets`, `strcpy`, `strcmp`, etc., were instrumented, as well as the individual applications. A summary of the dynamic characteristics of the test programs is given in Table 2.

1. Quicksort

The general quicksort algorithm takes a pointer to an array, lower and upper bounds, and a comparison function. Any function may be passed in as the comparison function. Ten thousand random numbers were sorted.

2. Fingerd

The finger daemon, `Fingerd`, responds to remote `finger` command requests. Version 5.3 was simulated, which contains the vulnerability used by a worm in November 1988 to cripple computers around the world. The worm used a classic buffer overflow attack. Buffer overflows are common errors, so we use `Fingerd` to represent a

class of applications that have had similar bugs. The bug in `Fingerd` is caused by the `stdio.h` routine `gets` rather than the program itself. Performance was measured by running the loop 100 times with different value inputs. (As expected, when we run `Fingerd` with an input that causes a buffer overflow, our hardware-enhanced monitor promptly flags it.)

3. Bridge

Bridge is a classical min-max branch-and-bound game search which plays bridge with full knowledge of each player’s cards. A branch and bound min-max search dynamically constructs a search tree. The full algorithm is not relevant; suffice to say that the program is deeply recursive with frequent pointer and array accesses to varied indices. This program was written for a contest in an undergraduate Artificial Intelligence course in 1996. When we ran the Safe Pointer monitor on the bridge code, it found an array access with an index of -1. Prior testing had not exposed this defect, but dynamic pointer-checking did.

4. MinSpan

MinSpan is a minimum spanning tree generator using Prim’s algorithm. To create the input graph, we began with 500 nodes and, given a desired connectivity, used a random number generator to determine whether there was an edge between each pair of nodes. If an edge was present, we used a random weight. A binary heap was used to store the nodes that had not yet been connected.

5. MatMult

MatMult is a sparse matrix multiply algorithm. Two 200 by 200 sparse matrices of randomly generated numbers are multiplied. The matrices have 1585 and 2381 entries.

7 Results

In this section, we provide cycle-level simulation results of our mechanisms incorporated into a next-generation microprocessor. Specifically, we examine a system based upon an 8-wide superscalar microprocessor. Our simulation infrastructure is based upon Sim3, an object-oriented derivative of SimpleScalar [11]. The system parameters are shown in Table 3.

Parameter	Value
CPU Clock	1 GHz
CPU Widget(f, d, c)	8 / 8 / 8
INT/FP ALUs	4 / 4
INT/FP MULT/DIVs	1 / 1
Load/Store Queue Size	64
Line size	64 bytes
L1 Access Time	2 cycles
L1 I-Cache	64K bytes
L1 D-Cache	64K bytes
L1 Miss Penalty	11 cycles
L2 Cache	512K byte
L2 Miss Penalty	68 cycles
HAT Access Time	1 cycles
HAT Cache	8K bytes
HAT Request Queue	64 words

Table 3: Simulation parameters

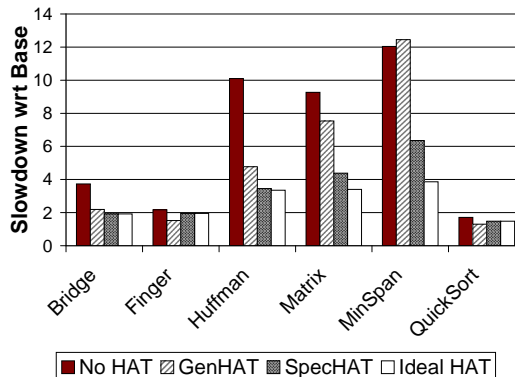


Figure 6: Execution times normalized to base program of software-only, GenHAT, SpecHAT, and an ideal SpecHAT.

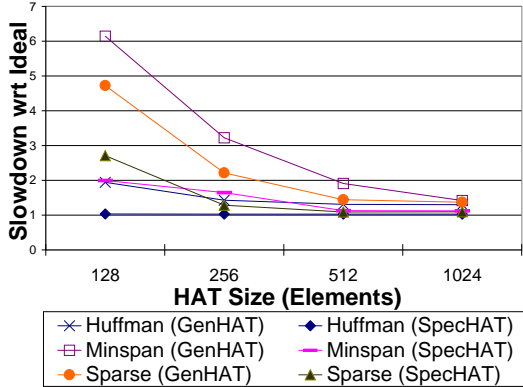


Figure 7: Execution times normalized to ideal SpecHAT for the GenHAT and the SpecHAT. As the size increases, performance approaches the ideal.

7.1 HAT Results

A major factor in software monitor overhead for safe pointers is the lookup of the pointer ID in a hash table to check whether the object is still alive. In Figure 6, we show the results of attacking this overhead using the HAT mechanism. The left bar corresponds to the baseline performance of inserting software checks. We then plot the generic HAT (GenHAT) and the specialized HAT (SpecHAT). Finally, we plot the runtime of the SpecHAT if all operations hit in the HAT, requiring no searches in memory. Note that this still includes the spatial checks, so it is expected to have a slowdown compared to the original program. The SpecHAT performs from 1.8 to 3 times better than software-only checking for applications suffering from at least a factor of four slowdown. Note that performance is very close to ideal in 4 out of 6 applications.

When the overhead is low, the GenHAT outperforms the SpecHAT. This is because of the interface used in the program. The SpecHAT requires two memory-mapped I/O instructions for each request, whereas the GenHAT only requires one specialized instruction. In addition, memory-mapped I/O instruction may not be executed in parallel and must be performed in order, slowing down instruction commits.

As the HAT misses increase, SpecHAT begins to outperform GenHAT for two reasons. First, searching for a missed entry takes more memory operations for GenHAT since each element takes twice the space. Second, with SpecHAT, instruction commit is only delayed if the hardware queue is full. With GenHAT, on the other hand, the instructions may not commit until the HAT operation itself is complete,

Operation: Find 128

	Cache Set Entries				Overflow Entries			
Before:	0	64	256	192	128	320	384	Empty
After:								
Inclusion:	128	64	256	192	128	320	384	0
Exclusion:	128	64	256	192	0	320	384	Empty

Figure 8: Example operation illustrating exclusion and inclusion algorithm behaviors. The cache and overflow memory state are shown before and after the operation.

and later instructions may be dependent on the GenHAT results. This makes SpecHAT more tolerant to burstiness, because the queue is based on throughput, not latency of individual instructions (until the queue fills up).

Figure 7 shows the performance of GenHAT and SpecHAT as the HAT capacity is increased. We give the capacity in elements because the size of each element differs for the GenHAT and SpecHAT. We depict the effects in the three applications with significant HAT cache misses. As the HAT size increases, we approach the ideal runtime. With 1024 elements, we observe a slowdown of at most 1.4. GenHAT responds best to the increase of HAT elements, since this reduces the overhead and memory operations incurred in high load applications. In SpecHAT, the size increase has a less pronounced, but still significant, effect. Our reference system has a capacity of 256 elements.

7.2 Overflow Algorithm

In this section, we describe an optimization that manages the chained aspect of our HAT structures to take advantage of temporal locality in key lookups. Unlike traditional caches, the location of a HAT element in memory is not fixed. When a writeback occurs, a convention must be followed for placing items in overflow space in memory. We investigated two algorithms for such evictions. Our first, the *inclusion algorithm*, is the traditional linked-list collision resolution for hash tables. It allocates cache lines as they are needed and places items in empty slots. When an item is found in the overflow memory, a copy of it is placed in the cache for quick subsequent accesses. The second, the *exclusion algorithm*, stores an element in only *one* of the HAT or the overflow memory. The difference between the two algorithms is depicted in Figure 8. We show the state before and after a “Find 128” operation. After the find operation, with inclusion, the

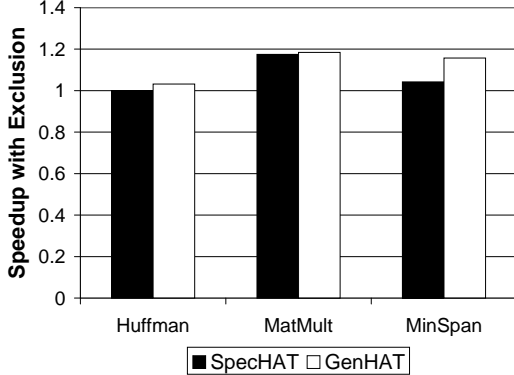


Figure 9: Performance of exclusion algorithm vs inclusion algorithm.

element 128 resides in both the HAT cache and the overflow memory. The performance results are shown in Figure 9. The exclusion algorithm achieves up to a 19% speedup over the inclusion algorithm.

The exclusion algorithm changes HAT behavior in three ways. First, every evicted element must be written back, whether it has been changed or not, since a copy no longer resides in the overflow memory. In the above example, with exclusion, if the HAT does not write back tag 128 when it is evicted, it will be lost. Second, when delete operations hit in the HAT cache, the operation is complete because it is guaranteed that no copy resides in the overflow memory. Third, and more importantly, temporal locality can be increased by writing back to the first empty slot in the chain of cache lines rather than the original location as in the inclusion algorithm. Notice that in the above example, the 0, the most recently accessed element in the overflow memory, is located in an earlier slot with exclusion than with inclusion.

The increase in write backs and early-terminated delete operations both affect what operations search through the overflow memory. Write backs require an extra search only if the original operation did not need to search the overflow memory, namely insert misses. Early-terminated delete operations only save searches for delete hits. Because stack ID's are short-lived, a high proportion of deletes are hits. Figure 10 shows the search rate with the inclusion and exclusion algorithms. The search rate is defined as the percentage of HAT operations that actually go to memory. If there is no overflow memory to search, then it is not counted as a search. We only display three applications, as the other applications had so few HAT

cache misses that the overflow algorithm did not affect application performance.

With the GenHAT, the increase in write back searches is offset by the decrease in delete searches, producing almost equal search rates. The SpecHAT, on the other hand, observes an increase in search rate with the exclusion algorithm. Specialization allows an optimization for delete hits with inclusion. An element that is in the HAT cache in the modified state when a delete request occurs may terminate without searching the overflow memory. The SpecHAT receives inserts for each element only once, meaning an element is only dirty once - when it is originally inserted. Thus it will not be in the overflow memory. The net result is an *increase* in search rate with the exclusion algorithm with the SpecHAT.

The large performance gain comes from the increase in locality by writing back items to the first available empty slot. Figure 11 shows the average number of L2 cache lines accessed per search with the inclusion and exclusion operations. The number of lines searched decreases with exclusion and the GenHAT requires more lines because the elements take twice as much space.

8 Discussion

Our results are encouraging, since a very small investment in hardware results in significantly accelerated pointer checking. The overhead associated with temporal checks has been significantly reduced. Our experience, however, also indicates that it will be very difficult to get further performance gains without abandoning conventional commodity microprocessor designs.

First, recall that our results are close to ideal for many applications. What remains is the overhead of the instructions inserted for instrumenting the code. These instructions can not be avoided unless the processor core is significantly modified. Some applications are somewhat sensitive to the cache memory inside the HAT, but this is still the same basic design and only a decision in resource investment.

Second, parallelizing the work is difficult. We performed a preliminary study [12] which explored the generic HAT with combined liveness and spatial checks and began to explore parallelization. Our current study, however, separates these checks and we discover that only the liveness checks would be worth parallelizing. We discovered that spatial checks are easy to parallelize, but they are so simple that communicating them outside the processor would take nearly as long as the work to be done. The liveness

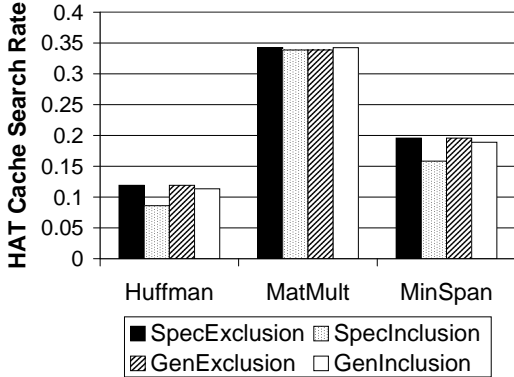


Figure 10: Search rate with the inclusion and exclusion algorithms.

checks, however, are difficult to parallelize because they are dependent upon allocations and deletions. Referring back to Table 2, we can see that the average number of liveness checks per table manipulation (allocation or deallocation) is small compared to the overhead necessary to synchronize, making parallelization of the checks of limited value.

Finally, parallelization may be possible if the checking tasks are completely migrated to monitoring processors or threads. This approach was taken in [13], in which a “shadow” thread on a separate processor computes both the control flow and the pointer checks for a program. Unfortunately, this approach can lead to significantly more work for the monitor than the original program. Coupled with our specialized hardware to reduce the work done by the monitor, shadow processing could provide performance necessary for run-time security, and we will explore this in the future.

9 Conclusions

As hardware speeds increase and costs decrease, software development is consuming an increasing proportion of overall system costs. Certain types of defects are very difficult to detect - for example, defects related to unsafe pointers in popular languages like C or C++. Dynamic pointer safety monitoring can effectively and quickly trap such errors, but entails a high performance penalty. In this paper, we presented a hardware technique that can be used to improve memory performance and lower performance penalties. Two versions of this technique were presented, a generalized hardware accelerated table (GenHAT) controlled through special instructions and a specialized HAT (SpecHAT) controlled through memory-mapped I/O instructions. While the GenHAT offers

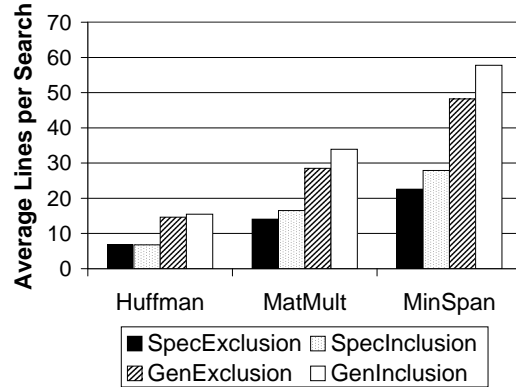


Figure 11: Average lines read per search with the inclusion and exclusion algorithms.

more flexibility, SpecHAT is cheaper in terms of space and design time since it makes no changes to the core and requires less data. Under a heavy load, it also outperformed the GenHAT.

The use of a simple hardware accelerated hash table can speed up pointer checking by about a factor of 2 with only a modest investment to hardware design. With the addition of sophisticated static analysis, the speed can increase even further. Our work continues with investigations of other hardware techniques, and other monitoring applications, such as security policy enforcement over untrusted binaries [14]. Finally, as hardware speeds increase and costs decrease, software development is consuming an increasing proportion of overall system costs. Thus, it is clear that the benefits in software engineering outweigh the hardware investment.

References

- [1] L. Cardelli, “Type systems,” in *The Computer Science and Engineering Handbook* (A. B. Tucker, ed.), pp. 2208–2236, CRC Press, 1997.
- [2] D. Wagner *et al.*, “A first step towards automated detection of buffer overrun vulnerabilities,” in *Network and Distributed System Security Symposium*, 2000.
- [3] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, “Efficient software-based fault isolation,” in *Proceedings of the Symposium on Operating Systems Principles*, 1993.
- [4] T. M. Austin, S. E. Breach, and G. S. Sohi, “Efficient detection of all pointers and array access errors,” in *Proc. Conference on Programming Language Design and Implementation*, 1994.
- [5] S. Kendall, “Bcc: run-time checking for c programs,” in *USENIX Toronto 1989 Summer Conference Proceedings*, 1983.
- [6] J.L.Steffen, “Adding run-time checking to the portable c compiler,” in *Software - Practice and Experience*, 1992.

- [7] Intel, Santa Clara, CA., *Introduction to the iAPX 432 Architecture*, 1981.
- [8] S. Inc, "Symbolics technical summary." <http://kogs-www.informatik.uni-hamburg.de/~moeller/symbolics-info/symbolics-tech-summary.html>.
- [9] N. Jouppi, "Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers," in *Proceedings of the 17th International Symposium on Computer Architecture*, (Seattle, WA), 1990.
- [10] L. Zhang *et al.*, "Pointer-based prefetching within the Impulse adaptable memory controller: Initial results," in *Proceedings of the ISCA-2000 Workshop on Solving the Memory Wall Problem*, June 2000.
- [11] D. Burger and T. Austin, "The SimpleScalar tool set, v2.0," *Comp Arch News*, vol. 25, June 1997.
- [12] S. for blind review
- [13] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in c programs," in *Software - Practice and Experience*, 1997.
- [14] D. S. Wallach, A. W. Appel, and E. W. Felten, "Safkasi: A security mechanism for language-based systems," in *ACM Transactions on Software Engineering and Methodology*, 2000.