Extending Commodity Microprocessors for Software Safety: An Experiment

Diana Keen, Fred Chong, Prem Devanbu, Matt Farrens
Computer Architecture Laboratory and Software Tools Laboratory
Dept. of Computer Science University of California
Davis, CA 95616, USA
{keend, chong, devanbu, farrens}@cs.ucdavis.edu

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. Our goal is to investigate approaches to exploit the increasing bounty of transistors in order to provide hardware structures that enhance software development, make debugging easier, increase reliability and provide run-time security. Unlike previous designs, we are specifically interested in "low-impact", evolutionary designs that conservatively extend current, commodity microprocessors. In this paper, we focus on run-time monitoring for pointer safety. We present new hardware designs aimed at boosting the performance of the best current (purely software) approaches, and evaluate the resulting performance gains. Our evaluations provide evidence of considerable acceleration and suggest some new opportunities for even greater gains. We also place our approach in the broad context of software safety, including both pure software approaches such as strong typing, sandboxing, proof-carrying code and software fault isolation, as well as traditional hardware-assisted approaches such as tagged architectures.

1. INTRODUCTION

For several decades now, advances in chip manufacturing technology have provided designers with an ever-increasing pool of available transistors. Designers have used 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.

Meanwhile, software development costs have skyrocketed. Large, complex software systems such as operating systems, databases, switching systems and desktop productivity applications are often expensive, behind schedule, and plagued with defects. In fact, maintenance and defect-removal costs are the major factor 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? This is the central focus of our work and this paper. We do not propose that hardware mechanisms will solve all our problems. Rather, we carefully apply hardware to accelerate dynamic techniques that augment static software techniques.

In this paper, we will address a specially nasty problem: 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 [4], and may not manifest itself until well after 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.

In type-unsafe languages like C and C++, memory access errors are caused by the incorrect use of pointers, and the only way to detect them immediately is via dynamic checking (which has a very significant performance impact). Strongly typed languages like Java can prevent these errors by means of static type-checking, combined with limited dynamic type checks. Unfortunately, because of legacy assets, efficiency, and available talent, it is likely that systems written in type-unsafe languages will remain under active development or maintenance for the foreseeable future.

Our primary goal is to accelerate dynamic checking in C and C++ code by providing architectural support. Dynamic checking will then become practical in more development settings, thus leading to easier, more effective defect isolation in a wider range of systems. As a case in point, during our experiments we *uncovered a previously unknown defect* in a medium-sized system. Dynamic monitoring may also become practical for some "live", operational systems in settings where safety is the dominant concern, and breaches must be detected immediately.

Our research has been animated by some key technical and

business observations about current and future processor architectures:

1. Current processor-core designs are complex and finely-tuned; changing the core will require extensive re-design and verification time. Consequently, we attempt to alter the core processor design as little as possible and place hardware outside the core. Adding tagged data-types, for example, would require extensive core re-design.
2. Placing monitoring tasks in the critical path of the execution of the main program will cause slow-downs. Therefore, we seek a design that allows monitoring to be performed in parallel with the main program.

With these observations in mind, we focused upon monitoring for a pervasive problem: protecting memory from improper use of pointers. Our experimental approach had three steps:

1. We evaluated existing, purely software-based dynamic pointer safety in order to identify specific tasks which would benefit from hardware-based acceleration.
2. We designed mechanisms to accelerate these tasks. A central goal in our design was to seek *low-impact* mechanisms that could be added to existing, commodity microprocessors. Our base architecture was a Simultaneous Multi-threaded machine, similar to the upcoming Hyper-Threading Pentium processor from Intel [5]. We seek low-impact in terms of performance, transistor usage, and on the design/layout of the existing microprocessor.
3. Finally, we evaluated the performance of our mechanisms (using simulation) over a set of applications with different characteristics. The results of this evaluation study suggest further opportunities for performance gains.

The paper begins in Section 2 with a brief summary of the historical context of this work. In Section 3, we present our software safety baseline system. We describe the state-of-the-art software implementations and analyze their performance bottlenecks. We then present an overview and design rationale of the acceleration mechanisms to address these bottlenecks in Section 4. The performance gains provided by these mechanisms (in simulation studies) are evaluated in Section 5. We conclude and look at future directions in Section 6.

2. HISTORICAL CONTEXT

There has been a great deal of work done on the broad topic of software safety. Software safety, from our perspective, is concerned with executing untrusted (defective or malicious) code in a manner that quickly detects and rejects actions that compromise protected resources. The precise context varies: the type of protected resource that might be of concern (kernel memory, other tasks' memory, file system, etc.), the trust model (mobile code, shrink-wrapped code, freeware etc.), the available information (safety proofs, source code, signatures, trust policies), and the required performance, are all key dimensions of the design space. A wide range of solutions have been developed; they can be broadly classified into software- and hardware-based solutions.

2.1 Software-based Safety

Strong typing, as used in languages such as Pascal, Java, Haskell, or ML, ensures that an object identified as a certain type is always of that type during execution. This avoids memory corruption. Static type-checking, augmented with dynamic checks (e.g., for array accesses) ensures this property. However, large volumes of legacy code written in untyped languages such as C and C++ will continue to demand investment in support and maintenance; it is unlikely that organizations will simply discard the hundreds of billions of dollars invested in such systems and rewrite them in strongly-typed languages.

Static analysis of source [19] 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. More recently, Xu *et al* [22] proposed a static analysis approach to checking type-safety of binaries; the scaleability of this approach is not yet clear. Static analysis is related to *Proof-carrying codes* [16], which can be automatically derived by a compiler, and provide binaries (sans source) which have *statically verifiable* properties; the goal is to provide evidence of safety without having to reveal the source. Unfortunately, such information-enriched binaries may risk revealing important intellectual property contained in the data-structure definitions and invariants[1].

Dynamic methods can detect errors such as array-bounds overflow or security policy violations at run-time. Approaches include source- or binary-instrumention [20], 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 approach, which has been quite popular [1, 14, 12], pointers record bounds information for an object. Each time a pointer is used, and the pointer value could have changed since the previous use, a check is performed to make sure it is still within the correct object. In addition, a check is performed to ensure that the pointed-to object has not been freed. Static analysis is performed to reduce the number of checks that must be done. Because of limitations in static analysis due to function calls and control branches, this approach generally entails substantial overheads. Conventional parallel implementations [17], which attempt to place the monitoring function in a "shadow" thread on a separate processor in a multi-processor machine, results in a slowdown of up to factor of ten or more. Through careful investment of silicon towards critical checking operations, however, we find that overheads can be substantially reduced.

2.2 Hardware-based Safety

[1]The structure of data is the key to understanding programs. See Brooks [3] or Raymond [18]

Numerous hardware approaches to accelerated run-time checking have been used, but they are either incompatible with commodity microprocessors or limited in capability. For example, the memory management system of a processor could be harnessed to provide pointer safety. One could allocate segments in segmented memory to confine pointers within the proper segment; the memory management hardware would then cause an interrupt when the pointer crosses a segment boundary. This would prevent some errors, since buffers would only be allowed to write into other buffers and could not corrupt the stack through normal string operations. It does not, however, stop them from overwriting each other; if different types of data are in the same segment, an unsafe pointer would not be prevented from corrupting data within its own segment. It also may not always be practical to allocate a separate segment for each type of data used in the program, and not all memory systems support segments.

Sun's Pico-Java for Java provides hardware support for an efficient stack, but no acceleration for run-time checking. The ambitious Intel iAPX432 [11] processor aimed to provide a rich set of run-time checks to support the ADA language. However, the powerful and extensive set of features provided for checking resulted in memory accesses that were unacceptably slow. Tagged architectures, such as the LISP-specific Symbolics [10], provided hardware support for 4 types (list, atom, integer and NIL) via 2 extra tag bits. This mechanism was also used implicitly for bounds-checking. Hardware tagging a language with an unbounded number of types is resource-intensive in terms of extra memory that is used and intimately affects the entire design of the processor. Changing a commodity processor such as the Intel Pentium to include tagging would be prohibitively expensive.

3. POINTER SAFETY

We take as a motivating application run-time checks to prevent illegal pointer references. In this section, we describe our approach to dealing with this application and the evaluation suite of programs used in our experiments.

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 delayed, mysterious, hard-to-trace software failures. 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 these types of mistakes are easily missed in testing.

3.1 Safe pointers implementation

For our study, we use an augmented pointer representation which allows pointer checking on commodity microprocessors. Specifically, we use the scheme proposed in proposed in [1], which is the best performing, easily repeatable, published implementation. Each pointer has associated information about the object to which it points. This "Safe Pointer" structure contains the current value of the pointer, 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 ensures that the value of the pointer is within the bounds of the object. A *temporal validity* check ensures that the referred object is still live, *i.e.,* it has not been freed, and perhaps re-allocated for a new instance of a different (or even the same) type. The information required for the spatial check is contained within the safe pointer datastructure. However, we must also determine temporal validity. For this, we maintain a separate temporal hash table that indicates whether an object with 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. Heap allocations are determined at run-time. The ID and base of both are obtained at run-time. For global objects, all the information is known at compile-time. As long as an object is active, its ID is entered in the temporal hash table; once the object is freed (upon explicit de-allocation for heap objects, and a stack pop for stack objects) the ID is expunged from the hash table.

The code must be instrumented to both maintain these datastructures and check pointer validity. Static analysis is performed on the source code to identify (conservatively) the relevant locations, and we then insert instrumentation to perform the following tasks:

- Update the hash table every time a heap pointer is freed, or allocated. In addition, we update on subroutine entry or exit.
- Whenever a pointer is assigned, update the associated safe pointer data.
- Perform checks each time a pointer is used.
- Handle unstructured transfers *i.e.,* `setjmp/longjmp` correctly (details are omitted for brevity).

To accelerate this basic mechanism, we first studied the performance of the purely software implementation, to understand where the bottlenecks were. This study was performed using a low-level hardware simulator to obtain detailed data. We used several programs to evaluate the base software performance.

3.2 Test Programs

We chose a variety of programs to determine the performance of our safety mechanisms. Our performance evaluations were obtained using cycle-by-cycle simulation. The

Application	Source Lines	Accesses	Assigns	Allocs
Quicksort	45	10	4	4
MinSpan	215	50	49	7
Fingerd	58	13	10	4
MatMult	11	3	0	3
Bridge	2000	224	51	28

Table 1: Static characteristics of test programs

simulator we used models all the functional units of the processor, including our novel mechanisms. Simulated functional units include (for example) instruction fetch units, branch prediction logic, pipelines, and caches. Because simulation of our novel hardware systems is several thousand times slower than real time, we experimented with small programs.

To evaluate our monitoring mechanisms, we used several programs that require dynamic run-time checks to guarantee security. Library calls such as `gets`, `strcpy`, `strcmp`, etc., were instrumented, as well as the individual programs. A summary of the static characteristics of the test programs is given in Table 1.

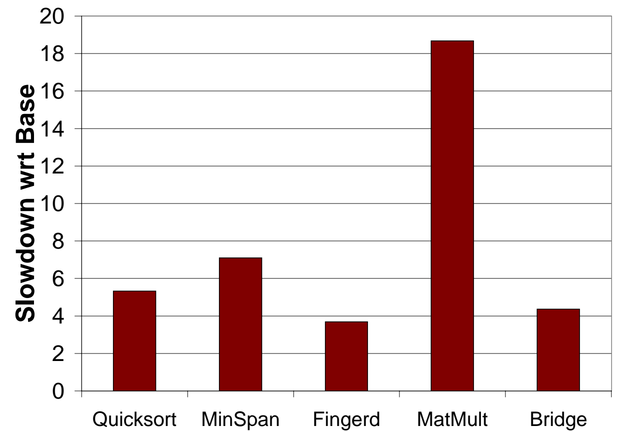

Figure 1: Slowdown induced by the safe pointer monitoring task. All our sample applications exhibit considerable slowdown. Matmult is the worst, because of frequent array accesses.

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 responds to remote `finger` command requests. We used Version 5.3, 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 programs 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 attack `Fingerd` with 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 min-max search (without the branch and bound) dynamically constructs a search tree. this is continued for a certain depth. 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 classic matrix multiply algorithm. Two 40 by 40 double-precision floating point matrices of randomly generated numbers are multiplied. This was selected because it has a very tight loop.

3.3 Baseline safe pointer workload profile

We instrumented the above programs (by hand) to perform pointer-safety monitoring. Elementary optimizations were performed to remove redundant checks in close proximity with no assignments in between. We then gathered data to help us identify aspects of the task that are candidates for hardware acceleration.

First, in Figure 1, we show the performance effect of inserting the monitoring code. Clearly, pointer safety induces a very heavy workload. Most applications suffer a substantial slowdown, ranging from about a factor of 4 to almost a factor of 20. Not surprisingly, the slowdown depends on the level of pointer or array usage, with the matrix multiply application generating the greatest amount of monitoring workload. To validate our safe pointers implementation, we compared MinSpan to the results in Austin et al[1]. The unoptimized slowdown is close to 8, and the optimized slowdown in [1] is about 6.25. Our result of 7 lies in between. This discrepancy will affect the results, as discussed in Section 5.

In order to accelerate this considerable workload, we attempted to determine where the monitoring effort was being spent. We conjectured that the bulk of the time was being spent on the table lookup (to determine the pointer lifetime/state). We ran three experiments in order to break the execution time down between the original program, the time spent in hash table lookups, and the rest of the monitoring overhead. We ran one program without any checks, another with full monitoring, and a third with hash table lookups

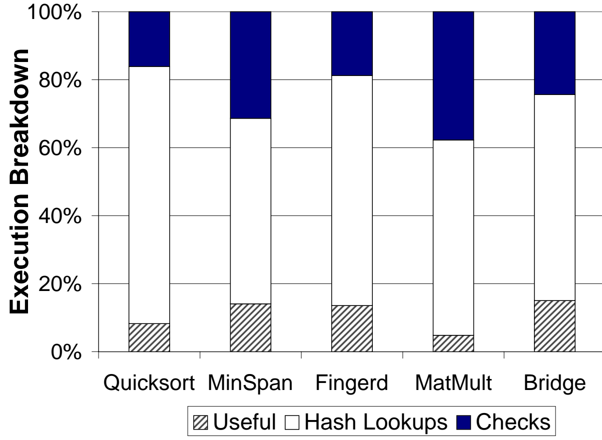

Figure 2: Workload breakdown of applications instrumented to perform dynamic pointer monitoring. The cross-hatched area is the application without monitoring. The white area is the hash lookup time; the rest is the time spent on checking. Note the high proportion spent on lookups; we seek to accelerate this

that were artificially given a latency of zero. From these, we were able to determine the proportion of time consumed by the actual lookups. The results are shown in Figure 2. The cross-hatched area is the original program time; the rest is the monitoring overhead. The largest (white) portion of the overhead is consumed by the hash lookup: this is where the hash datastructure that records the life-time of a pointer is consulted. The rest (black) portion is the other portion of the overhead in the safety computation. As a first step, we therefore chose to focus our hardware acceleration efforts on the hash table lookup workload. This led us to the design of the hardware accelerated table, or HAT.

In addition, we conjectured that during pointer safety monitoring, (because of the presence of such common programming idioms as loops, structure accesses and class member accesses) pointers in a very close spatial range will often be generated at about the same time. Therefore, it would be desirable to combine the checking of all these pointers into one single check. We seek to accelerate this mechanism as well; however, for continuity, we defer the discussion of this until Section 4.2. In the following section, we describe both of these mechanisms in more detail.

4. ARCHITECTURAL MECHANISMS

We propose two novel hardware mechanisms to support dynamic pointer checking in commodity microprocessors. Both mechanisms are specifically designed to have minimal impact on the design complexity, area, and performance of existing processors. In particular, performance is *not affected* when pointer checking is not used and performance is substantially enhance for pointer checking.

First, we propose a Hardware Accelerated Table (HAT) that accelerates hash-table lookups. The HAT is similar in functionality to other content-addressable memories in a processor (such as the TLB), but is software-accessible and implemented using substantially less expensive combination of parallel search hardware and conventional level-two cache.

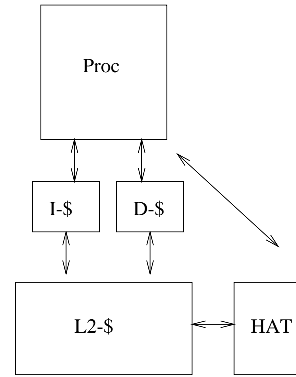

Figure 3: High-level hat design. I$, D$ and L2$ stand for instruction, data and L2 caches. The processor makes requests to the HAT, which stores its overflow entries into the L2 cache.

Second, we propose a mechanism that combines redundant pointer checks. Static analysis alone can not reduce all redundant checks because of the uncertainty introduced by branches and pointer aliasing. We use a hardware Reduction Queue (HRQ) mechanism to combine redundant requests at run-time. In order to take advantage of this queue, we must separate the pointer checking code and the original program into two separate threads. The checking thread can then read the combined requests from the HRQ; we will evaluate whether the queue mechanism compensates for the overhead of using an additional thread.

4.1 Hardware Accelerated Table

The HAT is essentially a cache for hash table lookups coupled with the logic to find elements that have been evicted from the cache. The concept of a hardware-supported lookup table is not new – TLBs are a good example of a common microprocessor component which provides such support. The HAT has three key distinctions, however. First, the HAT is accessible directly through software. Second, the HAT is is only 4-way set-associative, in contrast to the expensive, fully-associative hardware structures of a TLB. Third, when an element in the HAT is not found in its cache, we provide a simple engine that searches in memory for the element, as opposed to trapping the operating system to find the element for us. This allows the program to continue to run undisturbed.

The HAT will provide lookups, insertions, and deletions through a special instruction using a register-based interface:

```
HAT_lookup <key_register> <dest_register>
HAT_insert <key_register> <data_register>
HAT_remove <key_register> <dest_register>
```

Several issues influenced the design of the HAT:

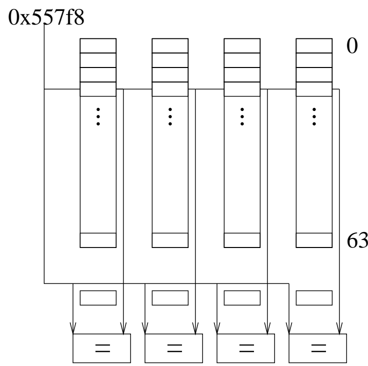

Figure 4: The HAT Architecture. We use a 4-way set-associative cache to accelerate recently-used tag lookups; with 64 sets, and 4 elements per set. We first index (in constant time) into one of the 64 sets; then all the elements of the set are read out, and compared in parallel. Hits are thus handled quickly (both update and lookup). Misses overflow into the memory sub-system, through the L2 cache.

- The size of the HAT is static
- The access time of the HAT is proportional to its size
- Placing the HAT 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.

To avoid adding delay into the tightly optimized core of our commodity microprocessor, we placed the HAT near the edge, next to the L2 cache. This makes the delay to and from the processor high, but also allows fast access to the large amount of storage in L2. Figure 3 shows the HAT its relationship to the chip components.

The HAT uses a 4-way set-associative memory (with a 4-element set size). There are 64 sets. This means that *all* possible unique ID's (UID) are mapped into one of 64 sets (buckets), with at most 4 elements per bucket stored in the HAT at any time. Upon lookup, we index into one of the sets in constant time, using 6 specific bits of the UID; this gives us 4 elements, which are compared simultaneously using the 4 comparators at the bottom (See 4). The elements in the HAT are treated as a cache - they are the most recently used items for that set entry. In the event of a miss, the associative set contains a pointer to the first element of a linked list (of sets of 4) in L2 cache memory. These buckets are searched automatically, without main processor intervention. The HAT relies on trap-generated system calls to allocate memory; it begins with a certain allocation, and generates calls for additional spillover memory as needed. We modeled this allocation workload in our simulations as additional average overhead.

There are some complications when a HAT miss spills over into the L2 cache. The L2 cache line size may be larger or smaller than the HAT set size. For maximum performance, we seek to fill up the HAT set each time, and perform all 4 comparisons in parallel. If the cache line is larger, we can perform several rounds of comparisons for each HAT cache-line read. If not, we have to read several cache-lines together to fill the HAT set. Our simulator had a smaller cache line; rather than simulating the HAT misses directly, we simply added additional overhead to the HAT access.

HAT operations are assumed to take 6 cycles in our results, distributed in the following manner—because we place the logic next to the L2 cache rather than in the processor core, we charge 1 cycle in wire delay to send the request to the HAT. In cycle 2, the HAT is searched. If the element is not currently in the HAT, the search continues from the L2 cache. We assume three cycles to access the L2 cache, transfer the line, and search the line. This is because of the close proximity to the L2 cache. If the entry is found in the first overflow line from the L2 cache, the result is returned in cycle 6. In reality, the average access time was between three and four cycles because most entries are found in the HAT and need not search in the L2 cache, but we used the conservative 6 cycle latency estimate to compensate for our small application sizes, and (infrequent) HAT misses.

The HAT must deal with speculation. Modern processors "speculatively" execute instructions that may be in an incorrect control path due to a mispredicted branch. In such cases, the processor must be able to rollback to the state before the offending instructions. This is accomplished by maintaining a type of "hardware checkpoint" of the memory accesses at the start of a speculative execution path, and tracking state along this path; this information is stored in the so-called "memory re-order buffer" (MRB) and is used when the path is committed or rolled back. In our case, rather than direct memory accesses, we need to checkpoint and track pointer-safety information that is stored by the hardware instructions accessing the HAT; recall that the HAT is an associative memory addressed by a unique pointer ID. Fortunately, there is a simple solution. Since HAT lookups and updates are just like memory loads and stores, the existing checkpoint/restart mechanism for memory can simply be reused. For this, we add one extra bit to each element of the MRB, and use this bit to distinguish between memory access and HAT accesses.

Since the HAT "by-passes" the memory hierarchy, and goes directly to the L2 cache (to reduce spatial layout impact on the processor and port restrictions in the L1 cache), one might wonder about potential conflicts with the L1 cache. However, this can only arise if the user code in the processor attempted to touch the HAT's own data structures; this is incorrect, and would be detected and prevented by the safety mechanism.

Finally, we sized the HAT using experiments. Empirically, we found that with 64 buckets, 90% of the queries lie within the first four elements; so we expect that our sizing is adequate. The HAT has 64 sets of four entries, each entry being 16 bytes long (two 8-byte words). Each set also has a single pointer (8 bytes) that points to the overfill location in memory. This means that each set has 72 bytes. The HAT contains logic plus 4608 bytes, or 4.5KB, of storage.

We can estimate the chip area of the HAT by comparing it to an L2 cache. The Pentium4 has a 256K on-chip L2 cache. The storage of the HAT is 1.7% of the storage of the L2 cache. The HAT also needs logic to fetch from the L2 cache and to decide when to stop looking for a match. We generously overestimated the logic needs, allocating 100,000 transistors, which is equivalent to 2KB of storage space. This means the area of the HAT is less than that used by 6.5KB of L2 cache, or 2.5% of the size of the Pentium4 L2 cache. Since the L2 cache consumes less than 16% of the area of the Pentium4 [9], the HAT requires less than .4% of the chip. As silicon densities increase and cache sizes reach diminishing returns, it is clear that a small investment in the HAT will be an attractive design decision.

The HAT is similar in spirit to victim caches [13] and has a similar goal as pointer-based prefetching [23]. However, the HAT takes an additional step in supporting comparison operations that provide the user an efficient, low-cost implementation of an associative memory. Associate memories occur in many different algorithms, and this mechanism could well provide useful acceleration for many applications.

4.2 Hardware Reduction Queue (HRQ)

Since the monitoring workload is so high, it would be clearly beneficial to reduce the number of checks that must occur. In this section, we describe the Hardware Reduction Queue (HRQ), a simple mechanism that provides dynamic elimination of redundant pointer checks.

4.2.1 HRQ Motivation

Informal, empirical observations indicate that many pointer checks can be combined into a single, equivalent event. While static analysis can remove some checks (for example, two accesses to the same pointer in a row without an assignment in between), others can only be detected at run-time. For example, consider the following code:

```
void bubblesort(T *A, int n, int (* cmp)(T, T))
{ int i;
    while(n > 0)
    {  for(i=1;i<n;i++) {
            if (cmp(A[i-1],A[i]) > 0)
            {   TYPE x;
                x = A[i];
                A[i] = A[i-1];
                A[i-1] = x;}
        }
        n--;
    }}
```

If A is a globally accessible, heap-allocated data structure, then something in the `cmp` function could deallocate it. Static analysis will be inhibited by both the variable offsets and the function call. So, a safety check is needed for `A[i]` and `A[i-1]` at run time. This identical check is repeated throughout the execution of the algorithm. If the compare does not deallocate anything (which is likely), then all of the checks to the same index of `A` are equivalent. At run-time, we can see that the same check is performed several times throughout the execution of the algorithm.

4.2.2 HRQ Structure

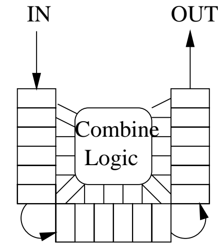

Figure 5: The HRQ (hardware request queue) structure. All the elements in the queue are constantly monitored by the combining logic in the middle which combines duplicate elements, thus reducing monitoring workload

Event1	Use	StackAlloc
Event2	Use	StackDealloc
Comparison Rule	Pointers identical	None
	Masked ptr vals same	
Events in between	No deallocs or copies	None Allowed
Action	Drop second event	Drop both events

Table 2: Rules for dropping events in Hardware Reduction Queue. If an instance of event1 and event2 pass each other that fulfill the comparison criteria and have no offending events in between, then the specified action can be taken.

To eliminate these redundant checks, we propose to add a queue which examines checks at run time and searches for checks to remove. To utilize this queue, however, we must split the monitoring tasks into a separate thread from the original program. This split establishes a producer-consumer relationship where the main program contains instrumentation which produces pointer checks and writes them to our Hardware Reduction Queue (HRQ). The HRQ eliminates redundant checks, and the monitor thread consumes the remaining checks by reading them from the HRQ. Reading and writing to the HRQ is achieved through normal loads and stores to special memory locations. In other words, the HRQ pretends to be a memory device. Multiple loads and stores are necessary to enqueue or dequeue a pointer check (one word at a time). This “memory-mapped interface” minimizes design complexity and verification costs because it is already part of commodity microprocessors.

To implement the HRQ, we borrow a concept from multiprocessor interconnection networks. The NYU Ultracomputer used a circular queue structure in each network router to combine messages to the same destination [7]. In that queue, messages can be combined as they pass each other in the queue if they have the same destination. Our HRQ, shown in Figure 5, has somewhat more complex rules for combining pointer checks, but has a similar structure. When we compare two events to see if we can drop them, we must be aware of what is in between the two events. For example, if we have two “Use” checks of the same pointer, and the pointers are exactly the same value, we might think that we can remove the second check. If there is a deallocation of that pointer in between, however, the second check is still necessary since it will find out the user is accessing an object

that has been freed. The rules are summarized in Table 2.

The HRQ will also be placed at the edge of the processor near the level two cache to minimize the impact on the processor design. To estimate the area required for the HRQ, we once again make a conservative comparison with an L2 cache. We will assume a 128 entry queue, which will have minimal impact on microprocessor chip area. Each entry is 8 bytes, which means the queue will store a total of 1K data. The HRQ logic is non-trivial, but will take substantially less than 250,000 transistors (the size of a small, embedded microprocessor), less than the area of 5K of L2 cache. This gives the HRQ a total area of 6KB L2 cache, about 2.4% of the Pentium4 L2 cache, or about 0.38% of the chip. Once again, this cost will become even less significant as chip densities increase and cache sizes reach diminishing returns.

As we shall see in Section 5, the HRQ can eliminate a substantial number of redundant checks. There is, however, a potential negative impact from separating the monitor into a second thread. In Austin's scheme[1], the information about a pointer was stored along with the pointer, and the main thread updated this information after the checks. This allowed the safety data to be retrieved along with the actual data in a single lookup. With a single application thread, there are no concurrency hazards on the safety information itself. If the monitoring task runs in a separate thread, we need to keep this information separate within the monitor thread to avoid concurrency hazards. This requires an extra lookup by the monitor. We return to this issue once we have presented the experimental data.

4.2.3 HRQ Reduction Through Padding

To further increase the effectiveness of the HRQ, we can trade-off (increasingly inexpensive) memory for checking workload by allocating memory in chunks of 128 bytes. If we do this, all accesses to that same 128 bytes are considered the same access, and can be combined into a single transaction.

Our previous code example strove to remove accesses to the *same* offset in an array. Here we attempt to remove accesses in *near* offsets in an array. Consider a tighter loop:

```
void mystrcpy(char *d, char *s)
{while(*s != '\0') {
                *d = *s;
                 d++;
                 s++; }}
```

In the above code, the values of s and d need to be checked each iteration because the value of the pointer changes each iteration. In such a tight loop, the pointer-checking workload quickly overwhelms the original program. If we add a mask to the check for uses that removes the lower bits of the pointer value, then we can combine any two uses that are to the same 128-byte section. In the above loop, if the string is more than 128 bytes long, we can get rid of 127 of 128 of the checks for each variable. Note that this does not stop the pointer from going off the edge of the array into the padding. We do allow extraneous accesses to the padding, but we do not allow corruption of any other data structures.

4.3 Security Model

Parameter	Value
CPU Clock	1 GHz
CPU Configuration	8-Wide
Line size	64 bytes
L1 Access Time	1 cycle
L1 I-Cache	64K bytes
L1 D-Cache	64K bytes
L2 Access Time	6 cycles
L2 Cache	1M byte
L3 Access Time	12 cycles
L3 Cache	4M byte
L3 Miss Penalty	62 cycles

Table 3: Simulation parameters

In this section, we specify our security model for our safety checks and architectural mechanisms. The model is simple. First we must guarantee that the input the monitor is receiving accurately reflects the behavior in the main process. For this to be true, we rely on the compiler or binary translator to correctly insert instrumentation to implement each safety mechanism. In the single threaded case, the pointer attributes can not be corrupted because accesses are checked before they are performed. In the case of a separate monitor thread, the two threads are actually different processes. Virtual memory hardware mechanisms ensure that two processes can not access each other's memory. This guarantees that no stray pointers in the main process can corrupt the monitor process' data structures.

5. RESULTS

In this section, we provide cycle-accurate simulation results of our mechanisms incorporated into a next-generation microprocessor. Specifically, we examine a system based upon the Alpha 21464 [15], Compaq's next-generation high-performance microprocessor. Intel's new "hyperthreading" technology [5], as manifest in the upcoming 3.4 Ghz Pentium processor, is a very similar architecture. Our simulation infrastructure is based upon SMTsim [6], a cycle-by-cycle simulator for a multithreaded Alpha 21464 that simulates functional units, branch predictors, and caches.. The system parameters are as shown in Table 3.

5.1 Results

How much do the HAT and the HRQ improve the baseline performance (seen in Figure 1) of the monitored application? We begin with Figure 6, where we see our primary result. By efficiently supporting table lookup and management functions, the HAT speeds up our pointer checking applications by 30-80%. The "No Hash" bar indicates the hypothetical performance if the HAT took zero time. The HAT has successfully removed 60-82% of the overhead associated with the hash table lookups. By placing the HAT search hardware out of the critical path of the processor core near the L2 cache, and by using the L2 cache for storage instead of a fully-associative memory, the HAT hardware delivers high performance at a fairly low increase in chip area and design complexity.

Next, we turn to Figure 7, where we evaluate the benefits of the HRQ in combining pointer checking operations to reduce the workload on the monitoring process. Except for MinSpan, we can see that pointer checking operations were substantially reduced, resulting in 50-80% reductions

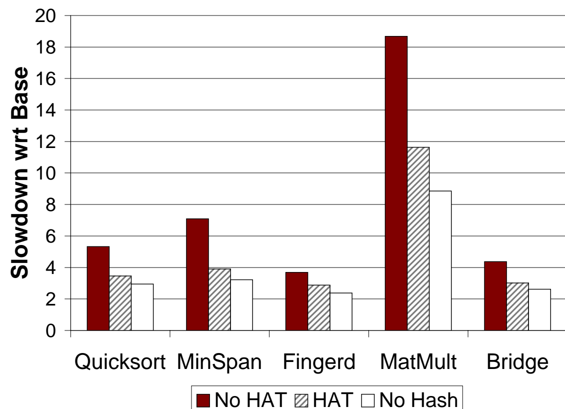

Figure 6: Performance of inserted code, accelerated by the HAT. For comparison, the "No Hash" version shows the hypothetical runtime if the HAT took zero time.

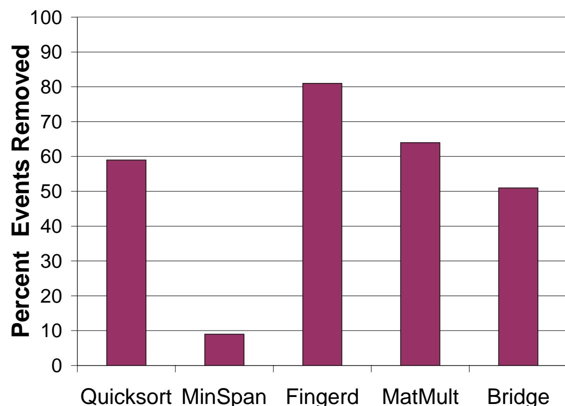

Figure 7: Percent of Monitoring Events Removed by hardware reduction queue

in workload. However, as can be seen in figure 8, the HRQ is not always beneficial, and even when it is, there is not much gain. It's notable that the *MinSpan*, with the fewest event removals, actually suffers the most. This phenomenon arises from the separation of the monitoring task into a separate thread: the spatial safety data is now separated into the monitoring thread, and requires an extra lookup, as explained at the end of Section 4.2.2. Our data thus provides useful evidence this this extra lookup dominates, even though many redundant monitoring events are removed.

Although these results are mixed, the workload reducing capabilities of the HRQ indicate a new direction. Microprocessor trends towards integrating multiple processors on a chip (Chip Multi-Processors or CMPs) [2] [8] suggest that monitoring tasks may be parallelized among several simple, inexpensive co-processors on a chip. The HRQ would be a valuable mechanism in such a setting, as parallelism overcomes the overhead of splitting the monitor from the main program. Our future work will investigate such designs.

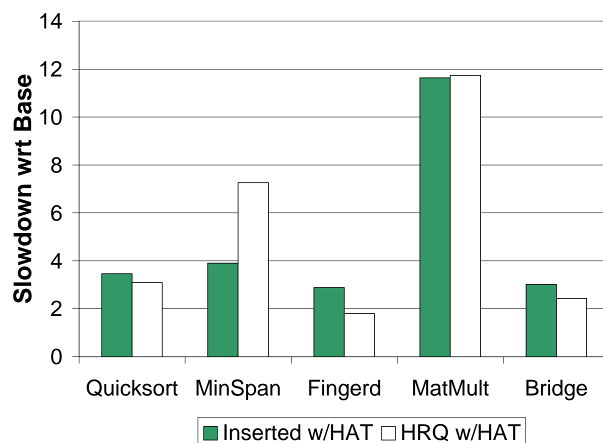

Figure 8: Performance of Safe Pointer Inserted Code and Hardware Reduction Queue, both using HAT

5.2 Implications

It is clear that our mechanisms provide a 40-70% reduction in overhead for pointer safety monitoring, and that will immediately benefit programmers. Several artificially seeded defects in different programs, corresponding to various types of unsafe pointer usage, were all detected as expected. In addition, *our system identified a previously unknown defect in the bridge program*, and it properly detected a violation and exited when we attacked the simulated `Fingerd` with a buffer overflow attack.

But what would be the cost of adding this feature to current chips? This would be quite modest for two reasons. First, because we can place the HAT quite far away from the processor core, the impact on the effort required to re-design and re-verify the processor core is modest. The HAT itself is very small and simple; we estimate that would take up only about 2.5% of the area of the L2 cache, and an even smaller amount of the next generation of chips. There is a modest impact due to add the control and data path for the

3 extra instructions—this would only affect the instruction decode functional unit and the commit unit.

Except for the highly array-intensive matrix multiply, the HAT-based acceleration runs at between 2x and 4x slow down. This is considerably better than the 3x to 7x slowdown without the HAT. We have removed an average of 75% of the hash table lookup overhead. With improved static analysis, we anticipate better performance. We believe that this makes the use of pointer safety monitoring more practical throughout development, thus enhancing the ability of programmers to isolate and eliminate a very difficult class of defects.

Our experience with the HRQ, although mixed at the moment, has exposed some important design considerations in the design of accelerators for all types of dynamic safety monitoring. While eliminating redundant events at run-time seems desirable, the accompanying need for multi-tasking could sometimes limit performance gains. The trade-offs must be carefully considered. We are exploring the use of multiple, smaller processors to support the monitoring task.

All in all, we believe that our results indicate that hardware support for safety monitoring such as the ones we have discussed above would be an attractive addition to commodity microprocessors. Since software costs are the increasingly dominant component of overall system costs, we believe that our overall approach has the potential to significantly *reduce* the overall cost of the system, based on savings in software development costs alone.

6. 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 debug - 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. We seek *low-impact* hardware designs that can reduce this penalty, but can still be added to commodity microprocessors with modest effort. In this paper, we tried two hardware techniques. A hardware accelerated hash table (HAT) can speed up pointer checking by more than a factor of 2. With the addition of sophisticated static analysis, the speed can increase even further. We also explore hardware queue combining, which currently shows mixed results, but which we believe has potential. Both hardware mechanisms are out of the critical path of the microprocessor and do not affect performance when not in use. Our work continues with investigations of other hardware techniques, and other monitoring applications, such as security policy enforcement over untrusted binaries [21].

7. ADDITIONAL AUTHORS

Jeremy Brown, Jenny Hollfelder, Xiu Ting Zhuang

8. REFERENCES

[1] 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.

[2] L. Barroso et al. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proc. International Symposium on Computer Architecture (ISCA 2000)*, 2000.

[3] F. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 2000.

[4] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[5] I. Corporation. Hyper-threading technology. http://developer.intel.com/technology/hyperthread/.

[6] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, September-October 1997.

[7] A. Gottlieb. The New York University Ultracomputer. In *Supercomputing*. University of California Press, 1986.

[8] L. Hammond et al. The Stanford Hydra CMP. *IEEE Micro*, March-April 2000.

[9] T. Hardware. Intel's new pentium 4 processor. http://www.tomshardware.com/cpu/00q4/001120/p4-03.html.

[10] S. Inc. Symbolics technical summary. http://kogs-www.informatik.uni-hamburg.de/ moeller/symbolics-info/symbolics-tech-summary.html.

[11] Intel, Santa Clara, CA. *Introduction to the iAPX 432 Architecture*, 1981.

[12] J.L.Steffen. Adding run-time checking to the portable c compiler. In *Software - Practice and Experience*, 1992.

[13] 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.

[14] S. Kendall. Bcc: run-time checking for c programs. In *USENIX Toronto 1989 Summer Conference Proceedings*, 1983.

[15] R. Merritt. Microprocessor forum: designers cut fresh paths to parallelism. *EE Times*, October 1999.

[16] G. Necula. Proof-carrying code. In *Proceedings of POPL 97*. ACM SIGPLAN, 97.

[17] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. In *Software - Practice and Experience*, 1997.

[18] E. Raymond and B. Young. *The Cathedral and the Bazaar : Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, 2000.

[19] D. Wagner et al. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, 2000.

[20] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles*, 1993.

[21] 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.

[22] Z. Xu, T. Reps, and B. Miller. Typestate checking of machine code. In *Proc. of ESOP 2001: European Symp. on Programming*, 2001.

[23] 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.