A Case for Unlimited Watchpoints

Joseph L. Greathouse[†] jlgreath@umich.edu
Hongyi Xin[‡ *] hxin@cs.cmu.edu
Yixin Luo[†§] luoyixin@umich.edu
Todd Austin[†] austin@umich.edu

[†]Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI, USA

[‡]SAFARI Research Group
Carnegie Mellon University
Pittsburgh, PA, USA

[§]UM-SJTU Joint Institute
Shanghai Jiao Tong University
Shanghai, China

Abstract

Numerous tools have been proposed to help developers fix software errors and inefficiencies. Widely-used techniques such as memory checking suffer from overheads that limit their use to pre-deployment testing, while more advanced systems have such severe performance impacts that they may require special-purpose hardware. Previous works have described hardware that can accelerate individual analyses, but such specialization stymies adoption; generalized mechanisms are more likely to be added to commercial processors.

This paper demonstrates that the ability to set an unlimited number of fine-grain data watchpoints can reduce the runtime overheads of numerous dynamic software analysis techniques. We detail the watchpoint capabilities required to accelerate these analyses while remaining general enough to be useful in the future. We describe a hardware design that stores watchpoints in main memory and utilizes two different on-chip caches to accelerate performance. The first is a bitmap lookaside buffer that stores fine-grained watchpoints, while the second is a range cache that can efficiently hold large contiguous regions of watchpoints. As an example of the power of such a system, it is possible to use watchpoints to accelerate read/write set checks in a software data race detector by nearly 9×.

Categories and Subject Descriptors B.3.2 [*Memory Structures*]: Design Styles—cache memories; C.0 [*General*]: hardware/software interfaces; D.2.0 [*Software Engineering*]: General—protection mechanisms; D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, testing tools

General Terms Design, Performance

Keywords Watchpoints, Data Race Detection, Deterministic Concurrent Execution, Taint Analysis, Demand-Driven Analysis

1. Introduction

Billions of dollars and millions of man-hours are spent each year attempting to build correct programs. As Bessey *et al.* stated, "Assuming you have a reasonable [software analysis] tool, if you run it over a large, previously unchecked system, you will always find bugs" [7]. The fact that developers looking to increase performance on multiprocessors must explicitly utilize concurrency only adds to this problem.

Numerous tools exist to help developers make more robust software. Valgrind's MemCheck, for instance, checks dynamic memory operations for errors such as memory leaks [37]. While such dynamic tools are powerful, they suffer from large computational overheads that limit their adoption to small groups of developers and dedicated testers. MemCheck can cause the original program to run 30× slower, while more heavyweight tools, such as Larson and Austin's symbolic execution engine, see slowdowns of over 200× [24]. More insidiously, because dynamic analyses can only observe bugs on executed paths, the power of these tools is further limited by slowdowns that reduce the number of situations observable in a reasonable amount of time.

Application-specific hardware is one way of solving this problem. Such mechanisms are custom-designed to accelerate individual analyses, but none offers a comprehensive way of accelerating many tools. As such, they are unlikely to meet the stringent requirements needed to be integrated into a modern commercial microprocessor; their benefits are too narrowly defined compared to their high area, design, and verification costs.

Commercial processors tend to favor generic solutions, where costs can be amortized across multiple uses. As an example, the performance counters available on modern microprocessors are used to find performance-degrading hotspots in software [39], but they are also used during hardware bring-up to identify correctness issues [48]. Similarly, while Sun added hardware into their prototype Rock processor in order to support transactional memory [10], the same mechanisms were also used to support runahead execution and speculative execution [11].

1.1 Contributions of This Paper

We theorize that a general acceleration mechanism should alert programmers when specified locations in memory are being accessed. In other words, having the ability to set a large number of data watchpoints (WPs) would benefit a wide range of analyses.

As the original program executes, many runtime analysis systems also perform actions on shadow values. In taint analysis, for example, each location in memory has a shadow value that marks it as trusted or untrusted. Checking this meta-data, in order to decide if analyses should occur, is slow. As an example, we previously demonstrated that checking read/write sets using software routines was 3-10× slower than using hardware to do the same [19].

[*] Author was at the University of Michigan and Shanghai Jiao Tong University while working on this project.

ASPLOS'12, March 3–7, 2012, London, England, UK.

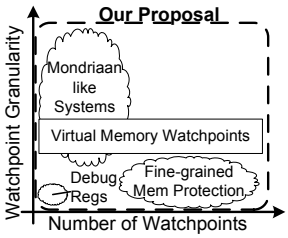


Figure 1: Existing Watchpoint Systems Are Inadequate. WP registers are too few in number. VM's granularity is too coarse. Mondriaan-like systems cannot quickly change many WPs, and other systems are often only useful for small regions.

This paper makes a case for the hardware-supported ability to set a virtually unlimited number of fine-grained data watchpoints. We will show that this mechanism can accelerate numerous software tools and is more general than the application-specific hardware often touted in the literature.

This is an improvement over existing memory-watching mechanisms, as qualitatively illustrated in Figure 1. Hardware watchpoint registers are too limited in number for the advanced systems we wish to accelerate. Virtual memory watchpoints, which are not constrained by hardware resources, are limited by the coarse granularity of pages. Finally, a number of proposals for fine-grained hardware memory protection and tagged memory systems exist in the literature, but they too are not general enough.

We present a mechanism that avoids these limitations. Our hardware/software hybrid watchpoint system stores per-thread virtual address watchpoints in main memory, avoiding hardware limitations on the total number of watchpoints. It makes use of two on-chip caches to hold these watchpoints closer to the pipeline. The first is a modified version of the range cache proposed by Tiwari *et al.* [40], which efficiently encodes continuous regions of watched memory. The second is a bitmapped lookaside cache, which increases the number of cached WP ranges when they are small.

By combining these mechanisms with the abilities to take fast WP faults and set WPs with user-level instructions, we can greatly accelerate tools that work on shadow data while remaining general enough to be useful for other memory-watching tasks. Our simulated results show, for instance, that a data race detector built using our technique can check read/write sets up to 9× faster than one built entirely with binary instrumentation and 3× faster than one using other fine-grained memory protection systems.

This paper presents the following novel contributions:

- We design hardware that allows software to set a *virtually unlimited number of byte-accurate watchpoints*.
- We study numerous dynamic software analysis tools and show how they can utilize watchpoints to *run more accurately and much faster*.
- We demonstrate that this design performs better on a wide range of tools and applications than other state-of-the-art memory monitoring technologies.

We detail the design of our watchpoint hardware in Section 2 and discuss software systems that could be built with it in Section 3. We compare our system to previous fine-grain memory protection works while running a variety of software analysis tasks in Section 4. Finally, we review other related works in Section 5 and conclude in Section 6.

Table 1: Hardware Watchpoint Support in Modern ISAs. Most ISAs support a small number of hardware-assisted watchpoints. While they reduce debugger overheads, their small numbers and reach are usually inadequate for more complex tools.

ISA	#	Known As	Possible Size
x86 [-64]	4	Debug Register	1, 2, 4, [8] bytes
ARMv7	16	Debug Register	1-8 bytes or up to 2GB using low-order masking
ePOWER	2	Data Address Compare	1 byte or 64-bit address with any bit masked or range up to 2^{64} bytes
Itanium	4	Data Breakpoint Register	1 byte to 64PB using low-order masking
MIPS	8	WatchLo/Hi	8 bytes, naturally aligned
POWER	1	DABR	1-8 bytes
SPARC	2	Watchpoint Reg.	1-8 bytes
z/Arch	1	PER	RO Range up to 2^{64} bytes

2. Fast Unlimited Watchpoints

This section describes a system that allows software to set a virtually unlimited number of byte-accurate watchpoints. We first review existing WP hardware and list what properties are needed to effectively accelerate a variety of software. We then present a design that meets these needs.

2.1 Existing HW Watchpoint Support

Watchpoints, also known as data breakpoints, are debugging mechanisms that allow a developer to demarcate memory regions and take interrupts whenever they are accessed [22, 27]. Using software to check each memory access can cause slowdowns, so most processors include some form of hardware support for watchpoints.

As Wahbe discussed, existing support can be broken down into specialized hardware watchpoint mechanisms and virtual memory [43]. The first commonly takes the form of watchpoint registers that hold individual addresses and raise exceptions when these addresses are touched. Unfortunately, as Table 1 shows, no modern ISA offers more than a small handful of these registers. This makes them difficult (if not impossible) to use for many analyses [14].

The second method marks pages containing watched data as unavailable or read-only in the virtual memory system. The kernel then checks the offending address against a list of watchpoints during each page fault. Though this system has been implemented on existing processors [4, 36], it has a number of restrictions that limit its usefulness.

Individual threads within a process cannot easily have different VM watchpoints. Additionally, the large size of pages reduces their effectiveness. Faults taken when accessing unwatched data on pages that also contain watched data can result in unacceptable performance overheads; we measured pathological cases on x86 Solaris that showed slowdowns of over 10,000×. Ho *et al.* also observed this problem and claimed that they "anticipate that using [finer-granularity] techniques would greatly improve performance" [21].

ECC memory can be used to set watchpoints at a finer granularity [32, 34]. By setting a value in memory with ECC enabled, then disabling ECC, writing a scrambled version of the data into the same location, and finally re-enabling error correction, it is possible to take a fault whenever a watched memory line is accessed. However, changing watchpoints, as well as taking watchpoint faults, is extremely slow in such a system. We do not explore this further.

In all, existing hardware is inadequate to support the varied needs of the wide range of dynamic analysis tools.

2.2 Unlimited Watchpoint Requirements

While this tells us what current systems do *not* offer, we must still answer the question of what a WP system *should* offer. Section 3 will detail watchpoint-based algorithms for numerous dynamic analysis systems, but in the vein of Appel and Li's paper on virtual memory primitives [2], we first list the properties which should be made available:

- **Large number**: Some systems watch gigabytes of data to observe program behavior.
- **Byte/word granularity**: Many tools use watchpoints at a very fine granularity to reduce false faults.
- **Fast fault handler**: Some applications take many faults, so this would greatly increase their performance.
- **Fast watchpoint changes**: Numerous tools frequently change WPs in response to the program's actions.
- **Per-thread**: Separate watchpoints on threads within a single process would allow tools to use watchpoints in parallel programs without taking false faults.
- **Set ranges**: Many tools require the ability to watch large ranges of addresses without needing to mark every component byte individually.
- **Break ranges**: It is also important to be able to quickly remove sections in the middle of ranges without rewriting every byte's WP. This is often used to carve out a working set of unwatched data.

2.3 Efficient Watchpoint Hardware

This section describes a hardware mechanism that operates in parallel with the virtual memory system in order to deliver on these requirements. Each watchpoint is defined by two bits that indicate whether it is read- and/or write-watched. To allow a ***large number*** of these watchpoints, the full list of watched addresses is stored in main memory. Accessing memory for each check would be prohibitively slow, so virtual addresses are first sent from the address generation unit (AGU) to an on-chip WP cache that is accessed in parallel to the data translation lookaside buffer (DTLB), as shown in Figure 2.

To deliver these watchpoints at ***byte granularity***, the cache compares each virtual address that the instruction touches and outputs a logical OR of their watched statuses. This check need not complete until the instruction attempts to commit, and so it may be pipelined.

If the WP unit indicates a fault, a precise exception is raised upon attempting to commit the offending instruction. This could either be a user-level fault, which is treated as a mispredicted branch, or a kernel fault. We assume the former in order provide a ***fast fault handler***. Using such handlers for kernel-controlled watchpoints, or cross-process watchpoints, is beyond the scope of this paper.

In order to yield ***fast watchpoint changes***, it is important that the on-chip caches be able to hold dirty data and only write back to main memory on dirty evictions. This, and the fact that the watchpoints are stored as virtual addresses, means that watchpoints can be changed with simple user-level instructions instead of system calls.

In order to support ***per-thread*** (rather than per-core) watchpoints, any dirty state in the WP cache will necessarily be part of process state. This can be switched lazily, only being saved if another process utilizing watchpoints is loaded, to increase performance. Additionally, each core will require a thread ID register so that individual threads can be targeted with watchpoint changes.

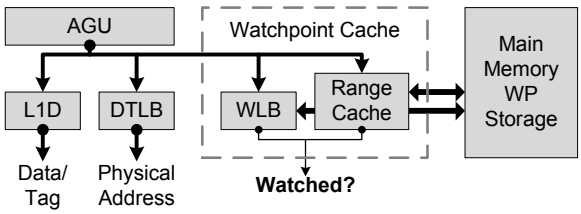


Figure 2: Watchpoint Unit in the Pipeline. Because watchpoints are set on virtual addresses, the WP system is accessed in parallel with the DTLB. This also allows WPs to be set with user-level instructions while not affecting the CPU's critical path.

Watchpoints are stored on-chip in three different forms. They are first stored in a range cache (RC), which holds the start and end addresses of each cached watchpoint and status values that hold each range's 2-bit watchpoint. This system, which is further explained in Section 2.3.1, helps support ***setting and breaking ranges***.

Small ranges can negatively impact the reach, or the number of addresses covered by all entries, of the RC. In this case, the WPs within a contiguous region of memory can be held in a single bitmap stored in main memory, rather than as multiple small ranges. The RC then stores the boundary addresses for the entire bitmapped region, and the status bits associated with that entry will point to the base of the bitmap. Accessing this bitmap would normally require a load from main memory, so we also include a Watchpoint Lookaside Buffer (WLB) that is searched in parallel with the RC. This is detailed further in Section 2.3.2.

Finally, because creating bitmaps can be slow (requiring kilobytes of data to be written to main memory), it can be useful to store small bitmaps directly on chip. By using the storage that normally holds a pointer to a bitmap, it is possible to also make an On-Chip Bitmap (OCBM) that stores the watchpoints for a region of memory that is larger than a single byte but smaller than a main memory bitmap. This is described in Section 2.3.3.

2.3.1 Range Cache

The first mechanism for storing watchpoints within the core is a modified version of the range cache proposed by Tiwari *et al.* [40]. They made the observation that "many dataflow tracking applications exhibit very significant range locality, where long blocks of memory addresses all store the same tag value," and we have found this statement even more accurate when dealing with 2-bit watchpoints rather than many-bit tags. This cache, shown as part of Figure 3, stores the boundary addresses for numerous ranges within the virtual memory space of a thread as well as the 2-bit watchpoint status associated with each of these regions.

Virtual addresses are sent to the RC in parallel with DTLB lookups, and the boundary addresses of each access are compared with those of the cached ranges. When any address that a memory instruction is attempting to access overlaps with a range stored in the RC, the hardware checks that range's watchpoint bits. If an overlapping range is marked as watched for this type of access, the instruction is set to cause a watchpoint fault when it commits. This will cause execution to jump to a software fault handler.

If there are no watchpoints set on a region of memory, the RC will hold a region with R- and W-watched bits both '0'. This means that if the range cache *misses* on any lookup, it should attempt to retrieve that range from main memory.

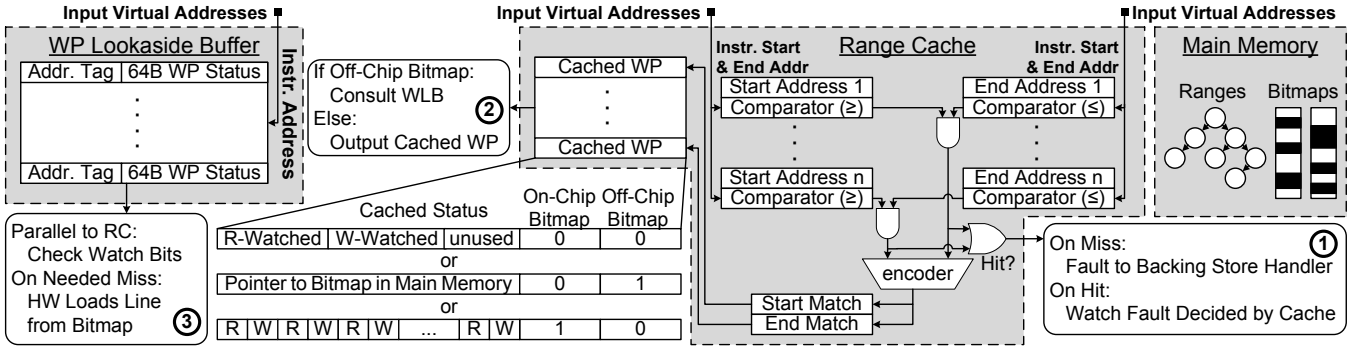


Figure 3: Unlimited Watchpoint Architecture. This system uses a combination of a range cache, center, with bitmaps and a lookaside buffer, left, to accelerate accesses to watchpoints that are stored into main memory, right. **(1)** On a RC access miss, a software handler loads new ranges from main memory. A hit causes the watchpoint system to check the associated status entry. **(2)** This range could be a uniform watchpoint, an off-chip bitmap, or an on-chip bitmap. In the case of an off-chip bitmap, the output of the lookaside buffer, which is accessed in parallel to the RC, is consulted. **(3)** If the WLB misses, the pointer from the RC status entry is used by the hardware to load a line in from the main memory bitmap.

The original RC design brought in 64 byte chunks from a two-level trie whenever it missed. We found this method inefficient for our tools, as it required a large number of writes to save non-aligned ranges. Instead, our RC causes a fault to a software handler on a miss. This handler loads an entire range from the storage system in main memory. We implemented a backing store handler that keeps a balanced tree of non-overlapping ranges, which was modeled after the watchpoint data structure in the OpenSolaris kernel.

Programs set watchpoints on their own memory space using range-based instructions, which are described in Section 2.3.4. These instructions can set or remove ranges by directly inserting the new watchpoint tag into the range cache, which uses a dirty-bit write-back policy to avoid taking a fault on every WP change. The RC employs a pseudo-LRU replacement policy, which keeps track of the most-likely candidate for eviction. If the range cache overflows and the LRU entry is dirty, the cache will cause a user-level fault that reroutes execution to the software backing store handler.

Updating a watchpoint range is more complicated than setting or removing, as it may require loading in ranges from the backing store to find the value that is to be modified.

2.3.2 Bitmap and Watchpoint Lookaside Buffer

The RC is designed to quickly handle large ranges of watchpoints, and its write-back policy reduces the number of writes to main memory. However, its reach can be limited if it contains many short ranges. In the worst case, a 128-entry RC, which takes up an area roughly equal to 4KB of L1D, may only have a reach of 128 bytes. We utilize a second hardware structure to handle these cases.

Bitmaps are a compact way of storing small watchpoint regions, because they only take 2 bits per byte within the region, as demonstrated in Figure 4. Instead of storing the start and end address for each small range, we therefore choose to sometimes store into the range cache the first and last address of a large region whose small watchpoints are contained in a bitmap in memory. This increases the required amount of status storage in the range cache, which originally only held the read-watched and write-watched bits. It instead requires 32 or 64 bits of storage to hold the pointer to the bitmap and one bit to denote whether an entry is a bitmap pointer or a range. The two watched bits can be mapped onto the pointer bits to save space.

We found it difficult to design a hardware-based algorithm to decide when to change a collection of ranges into a bitmap. Doing so in an intelligent manner requires knowing how many watchpoints are contained within a particular area of memory. This knowledge may best be gathered by the backing store software, as it can see the entire state of a process's watchpoints. We therefore leave it to the range cache miss handler to decide when to toggle a region of memory between a bitmap and ranges. The algorithm modeled in this paper moves a naturally aligned 4KB region from a range to a bitmap if the number of internal ranges exceeds an upper threshold of 16. The dirty eviction handler changes a bitmap back to ranges if this number falls below 4. This is illustrated in Figure 5.

While this mechanism increases the reach of the range cache, it could adversely affect performance if every access to a bitmapped range required checking main memory. This is especially true because WPs are set on virtual addresses, meaning that each access to the bitmap would wait on the TLB. In order to accelerate this process, our system includes a Watchpoint Lookaside Buffer (WLB). This cache is accessed in parallel to the RC, and any watchpoint status it returns is consulted if the RC indicates that this location is stored in a bitmap. On a miss in the WLB, a 64-byte line of WP bits is loaded by a hardware engine from the bitmap pointed to by the RC entry. Changes to the watchpoints in a bitmapped region cause a WLB eviction.

This WLB could be replaced with extended cache line bits, such as iWatcher uses to store its bitmapped watchpoints [50]. Sentry's power-saving method of only storing unwatched lines in the L1 (and thus only checking the WLB on cache misses) may also be useful [38]. We leave a more in-depth analysis of these tradeoffs for future work, and focus on a system with a separate watchpoint cache, much like MemTracker uses [42].

If a bitmapped range is evicted from the RC to main memory, the modeled software handler stores the entire range as a single entry in the balanced tree, along with the pointer. It then brings the entire bitmap range back into the range cache on the next miss, though it may also need to update the bitmap to handle changes that occurred in the range cache but have not yet been evicted.

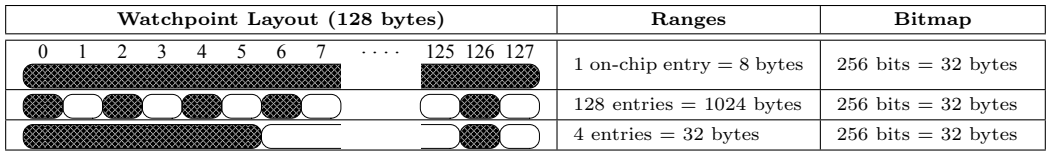


Figure 4: Different Watchpoint Storage Methods. This shows three examples of watched ranges. In the first, a range cache can hold the entire region in a single on-chip entry. The second, however, would take 1KB of on-chip storage if it were held in a range cache, while a bitmap method would only take 32 bytes. The final entry shows a break-even point between the two methods.

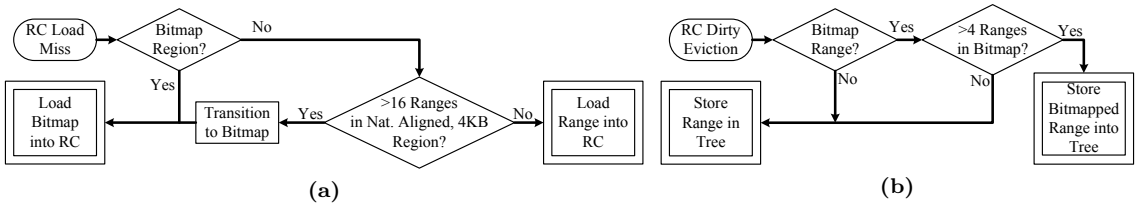


Figure 5: Software Algorithm for Bitmapped Ranges. Because hardware has a myopic view of the global status of watchpoints, we rely on the software fault handlers to choose when a region of watchpoints should toggle between bitmap and range.

2.3.3 On-Chip Bitmap

There are also situations where a collection of small ranges within a large region prematurely displace useful data from the range cache. Though the backing store handler may later fix this by changing the region to a bitmapped one, we also devised a mechanism to better utilize the pointer bits in a range cache entry when it is not bitmapped. It is possible to store a small bitmap for a region $\leq$16 bytes (on a 32-bit machine) within these bits, allowing the RC to slightly increase its reach. This on-chip bitmap (OCBM) range is stored much like a normal range, with an arbitrary start and end address. However, unlike a uniformly tagged range, the watchpoint statuses of OCBM entries are stored as a bitmap in the range cache's status tag bits, which normally hold either a pointer to a main memory bitmap or the 2-bit tag of a range. The low-order bits of the address are used to index into this bitmap, allowing a single range cache entry to hold up to 16 consecutive small ranges.

Unlike the bitmaps stored in main memory, the transition to OCBM requires little knowledge. Simply put, if a range becomes small enough to be held in a single OCBM entry, it is converted to one by the hardware. Returning to a uniform range will occur when the hardware detects that all of the watchpoints within the OCBM are equal. If an OCBM is chosen to be written back to memory, our backing store handler will store it as individual ranges. If two OCBMs cover adjacent ranges, the hardware will merge them only if their total size would still result in an OCBM.

2.3.4 ISA Changes

Interfacing with this watchpoint system requires modifications to the ISA. For instance, we must add instructions that can set or remove ranges from the RC, as well as instructions that allow the backing store handler to directly talk to the RC hardware. Table 2 lists the instructions that must be added and describes what each does.

The most complicated instruction semantics relate to modifying watchpoints in multi-threaded programs. The WLB, for example, must be able to respond to shootdown requests in order to remain synchronized across multiple threads. Most importantly, instructions must exist to add, remove, and update ranges of watchpoints for sibling processors. To do this quickly, our system must send these requests without going through the OS. One method of doing this involves broadcasting to all cores a process ID (e.g., the CR3 register in x86) and thread ID along with the request to perform a global update. Each core can then update its own cache entry if it matches the target process and thread ID. This instruction must be a memory fence, however, to maintain consistency between watchpoints and normal requests to watched memory locations. After the remote threads update their watchpoint cache, they must also send back acknowledgments. If any target thread ID does not return an acknowledgment within some timeout period, it may not be running, and its watchpoints may be saved in memory. The source processor must then raise an interrupt and allow the OS to update the unscheduled process's watchpoints.

3. Watchpoint Applications

This section analyzes potential applications for this watchpoint system. We detail how each requires some subset of the requirements listed in Section 2.1 and then develop watchpoint-based algorithms that can accelerate each analysis. Space limitations prevent us from detailing more than what we tested in our experiments, but Table 3 covers other tools, which are also briefly discussed in Section 5.

3.1 Dynamic Dataflow Analysis

Dynamic dataflow analyses associate shadow values with program data, propagate them alongside the execution of the program, and perform a variety of checks on them to find errors. This meta-data can represent myriad details about the associated memory location such as trustworthiness [12], symbolic limits [24], or identification tags [29]. Unfortunately, these systems suffer from high runtime overheads, as every memory access must first check its associated meta-data before calculating any propagation logic.

Ho *et al.* described a method for dynamically disabling a taint analysis tool when it is not operating on tainted variables, allowing the majority of memory operations to proceed without any analysis overheads [21]. Pages that contain any tainted data are marked unavailable in the virtual memory system. Programs will execute unencumbered when operating on untainted data, but will cause a page fault if they access data on a tainted page. At this point, the program can be moved into the slow analysis tool.

Table 2: ISA Additions for Unlimited Watchpoints. Watchpoint modifications must be memory fences, and instructions working on remote cores cause a shootdown in the remote WLB. Instructions used by the backing store handler can set all bits in a RC entry, and are also used during task switching.

Watchpoint Modifications	
`set_local_wp` *start, end,* {*r,w,rw,0*}	Adds a R/W/RW/not-watched range into this CPU's RC. Overwrites any overlapping ranges.
`add/rm_local_wp` *start, end,* {*r,w*}	Updates an entry in the RC of this CPU. If anything between *start* and *end* is not in the RC, it must be read from the backing store to properly update it.
`set_remote_wp` *start, end, tid,* {*r,w,rw,0*}	Adds an entry into the RC of the CPU with TID register *tid*.
`add/rm_remote_wp` *start, end, tid,* {*r,w*}	Updates an entry in the RC of the CPU with TID register *tid*.
Backing Store Handler	
`read_rc_entry` *n*	Reads the n^{th} entry of the RC (LRU order). This allows the oldest entries to be sent to the backing store. A bulk-read instruction could be used for task-switching.
`store_rc_entry` *start, end, status_bits*	Allows writing an entire range (including bitmap and OCBM status bits) into a range cache entry. Used for reading in a watchpoint on a RC miss.
Watchpoint System Interface	
`enable/disable_wp`	Enable or disable the watchpoint system on this core. Kernel-level instruction.
`set_cpu_tid` *tid*	Set the TID register on this CPU at thread switch time.
`enable/disable_wp_thread` *tid*	Toggle WP operation for any core with the same PID as this core's and TID equal to *tid*.
`set_handler_addr` *addr*	Set the address to jump to when a watchpoint-related fault occurs. If faults are user-level, then this instruction is user-level.
`get_cpu_tid`	Find the value in this processor's TID register.

Ho *et al.* discussed the problem of *false tainting*, where the relatively coarse granularity of pages causes unnecessary page faults when touching untainted data. Ideally, such a demand-driven taint analysis tool would utilize byte-accurate watchpoints. Additionally, although they partially mitigated the slowdowns caused by the lack of a fast fault handler by remaining inside their analysis tool for long periods of time, this can limit performance. Their tool conservatively remains enabled while performing no useful analysis.

A WP-based algorithm for this type of system can be summarized as setting RW watchpoints on any data that is tainted and running until a WP fault occurs. The fault indicates that a tainted value is either being written over or read from, and the propagation logic or instrumented code should be called from the WP handler. The tool should also remain enabled while tainted data exists in registers, as those values cannot be covered by watchpoints.

3.2 Deterministic Concurrent Execution

Deterministic concurrent execution systems attempt to make more robust parallel programs by guaranteeing that the outputs of concurrent regions are the same each time a program is run with a particular input [5]. This can be accomplished by allowing parallel threads to run unhindered when they are not communicating, but only committing their memory speculatively. If one thread concurrently modifies the data being used by another, they must be serialized in some manner.

Grace, one example of this type of system, splits fork/join parallel programs into multiple processes and marks potentially shared heap regions as watched in the virtual memory system [6]. Any time the program reads or writes a new heap page, a watchpoint fault is taken, whereupon the page is put into that process's read or write set and marked as read-only or available, respectively. As the processes merge while joining, any conflicting write updates cause one of the threads to roll back and reexecute.

While Berger *et al.* demonstrated the effectiveness of this system for a collection of fork/join parallel programs, using virtual memory watchpoints in such a way limits the applicability of their system. First, it only works on programs that can easily be split into multiple processes, which can lead to performance and portability problems in some operating systems. Additionally, it can only look for conflicting accesses at the page granularity. While many developers work to limit false cache line sharing between threads, it is much less likely that they care to limit false sharing at the page granularity. Such a mismatch can lead to many unnecessary rollbacks, again reducing performance.

Our watchpoint-based deterministic execution algorithm is similar to Grace, except that it works on a per-thread basis and sets watchpoints at a 64B cache line granularity. This could also be done at the byte granularity, but with higher overheads.

3.3 Data Race Detection

Unordered accesses to shared memory locations by multiple threads, or data races, can allow variables to be changed in undesired orders, potentially causing data corruption and crashes [31]. Dynamic data race detectors can help programmers build parallel programs by informing them whenever shared memory is accessed in a racy way [35].

We previously showed a data race detection mechanism that operated on similar principles to demand-driven taint analysis, though it required the careful usage of performance monitoring hardware to observe cache events [19]. That system could run into performance problems due to false sharing and may miss some races due to cache size limitations, among other issues.

It is possible to build such a demand-driven analysis system using per-thread watchpoints. Similar to how Grace operates, all regions of shared memory are initially watched for each thread. As a thread executes, it will take a number of watchpoint faults in order to fill its read and write sets in a byte-accurate manner. In our implementation, reads that take a watchpoint fault remove the local read watchpoint and set a write watchpoint on all other threads (in order to catch WAR sharing), while writes completely remove the local watchpoint and set RW watchpoints on all other cores (to catch RAW and WAW sharing). Any time an instruction takes a WP fault, the software race detector can assume that it was caused by inter-thread data sharing, and should

Table 3: Applications of Watchpoints. This is a sample of software systems that could utilize hardware-supported watchpoints to perform more efficiently and accurately. High-level algorithms that focus on how such systems would interact with the watchpoint hardware are given for each, as well as an overview of the watchpoint capabilities that would be useful or needed for each algorithm.

Software System	High Level Algorithm	Large Number	Byte Granular	Fast Handler	Fast Changes	Per Thread	Set Ranges	Break Ranges
Demand-Driven Dataflow Analysis	Set shadowed values as RW watched. Enable analysis tool only on watchpoint faults.	X	X	X	X			
Deterministic Execution	Start with shared memory RW watched in all threads. On local access fault: Check for write conflict between threads. If so, serialize. Unwatch (R or RW, depending on access) cache line locally. Rewatch cache lines on other threads after serialization.	X				X	X	X
Demand-Driven Data Race Detection	Start with shared memory RW watched in all threads. On local access fault: Run software race detector on this access. Unwatch (R or RW, depending on access) address locally. Rewatch address on other threads.	X	X	X	X	X	X	X
Bounds Checking	Set W-watchpoints on canaries, return addresses, and heap meta-data.	X	X					
Speculative Program Optimization	Mark data regions as W watched in speculative and normal threads. On faults: Save list of modified regions (perhaps larger than pages). Mark page available in this thread. Compare values in modified regions when verifying speculation.	X				X	X	X
Hybrid Transactional Memory	At start of transaction, set local memory as RW watched. On local access fault: Save original value for rollback. Check for conflicts with other transactions. Unwatch (R or RW, depending on access) address locally Unwatch memory for this thread on transaction commit.	X		X	X	X	X	X
Semi-space Garbage Collection	During from-space/to-space switch: Mark all memory in from-space as RW watched to executing threads. Update dereferenced pointer to be consistent on faults.	X				X	X	

send the access through its slow race detection algorithm. A similar WP-based race detection method was recently demonstrated in DataCollider, though they are only able to concurrently watch four variables due to WP register limitations [16].

This tool is a prime example of the need to break large ranges, as the program initially starts by watching large ranges and slowly splits them to form a local working set.

4. Experiments

Any slowdowns caused by issues such as WP cache misses should not outweigh the performance gains a WP system offers software tools. To that end, this section presents experiments that evaluate the performance of our system and a collection of other fine-grained memory protection systems when they are used to accelerate dynamic analysis tools.

4.1 Experimental Setup

We implemented a high-level simulation environment using Pin [25] running on x86-64 Linux hosts. The software analysis tools were implemented as pintools and were used to analyze 32-bit x86 benchmark applications. These pintools communicated with a simulator that modeled the watchpoint hardware. It also kept track of events that would cause slowdowns. The overheads of events that were fully exposed to the pipeline (e.g. faults to the kernel) were calculated by multiplying the event count by values derived from running lmbench [28] on a collection of x86 Linux systems. These values are listed in Table 4.

Events that are not fully exposed, such as the work done by software handlers, were logged and run through a trace-based timing tool on a 2.4GHz Intel Core 2 processor running Red Hat Enterprise Linux 6.1. These actions are listed in Table 5. The runtime of the timing tool is recorded and used to derive the cycle overhead of such events.

While this setup is likely to have inaccuracies (due to, for instance, caches and branch predictors being in different states), it is still useful in giving rough estimates of the performance of numerous different systems across a multitude of tools and benchmarks. It's worth nothing that even "cycle-accurate" simulators have inaccuracies compared to real hardware [47]. The larger testing space available to our fast simulation can still lead to useful design decisions.

4.1.1 Hardware Designs Modeled

To compare our design to previous works that could also offer watchpoints, we built models for a number of other hardware memory protection mechanisms. We assume that every system except virtual memory has user-level faults so as not to bias our results away from other hardware designs.

Virtual Memory – This models the traditional way of offering large numbers of watchpoints [2]. Touching a page with watched data causes a page fault, whereupon the kernel looks through a list of byte-accurate watchpoints to decide if this was a true watchpoint fault. If so, a signal is sent to the user code. If not, the page is marked available, the next instruction single-stepped, and the page is then marked unavailable again after the subsequent return to the kernel. The overheads of this system can be estimated as: (# true faults $\times$ ($T_{kernel} + T_{signal}$)) + (# false faults $\times$ $T_{kernel} \times 2$) + (# WP changes $\times$ $T_{syscall}$) + $T_{SWcheck}$ + T_{SWset} + T_{VM}.

MemTracker – This implements a design that has a lookaside buffer that is separate from the L1D cache (called

Table 4: Exposed Latency Values. These events are counted in our simulator, and each event is estimated to take the listed number of cycles, on average.

Event	Added Cycles	Symbol
Kernel fault	700	T_{kernel}
Syscall entry	400	$T_{syscall}$
Signal from kernel	3000	T_{signal}
User-level fault time	20	T_{user}

the "Taint L1" in FlexiTaint [41] and "State L1" in MemTracker [42]). In these tests, the State L1 (SL1) was a 4KB, 4-way set associative cache with 64-byte lines. The original MemTracker did not have user-level faults, so their backing store was a bitmap in main memory. Our initial tests showed that the vast majority of time was spent writing data to this bitmap, so we changed the system to use a software-controlled backing store handler similar to our design. The overheads of this system can be estimated as: ((# of faults + # SL1 misses) $\times$ T_{user}) + $T_{SWcheck}$ + T_{SWset}.

Mondriaan Memory Protection – MMP utilizes a trie in main memory to store the watchpoints for individual bytes [46]. Upper levels of the table can be set to mark large, aligned regions as watched in one action. The protection lookaside buffer (PLB) is 256 entries and can hold these higher-level entries (using low order don't-care bits). We did not utilize the mini-SST optimization because Witchel later described how such entries can yield significant slowdowns if permission changes are frequent [45]. The overheads for this system are: (# of faults + T_{user}) + $T_{HWcheck}$ + T_{MLPT}

Range Cache – This system is a 128-entry range cache, where each entry holds 2 bits of WP data. Tiwari *et al.* estimated that a 128-entry range cache with 32-bit entries would take up nearly the same amount of space as 4KB of data cache [40]. The OH is: ((# of faults + # RC misses + # write-backs) $\times$ T_{user}) + (complex range updates $\times$ 64 cycles) + $T_{SWcheck}$ + T_{SWset}.

RC + Bitmap – Our technique adds bitmapped ranges, OCBMs, and a 2-way set-associative, 2KB WLB to the range cache design. The size of the RC is reduced to 64 entries because of the extra area needed for these features. This system can have further overhead-causing events besides those of the range cache: ($T_{HWcheck}$ on WLB miss) + (time to decide to switch to/from bitmap ranges) + T_{bitmap}.

4.1.2 Software Test Clients

The software analyses discussed in Section 3 are utilized as clients for the simulated hardware watchpoint system. The overheads caused by the tools themselves are common amongst all watchpoint designs and are not modeled. In other words, the reported performance differences are relative to the meta-data checks only, not the shadow propagation, serialization algorithms, or race detection logic.

Demand-Driven Taint Analysis – This tool performs taint analysis on target applications, marking data read from disk and the network as tainted. As a baseline, we compare against two state-of-the-art analysis systems. The first, MINEMU 0.3, is an extremely fast taint analysis tool that utilizes multiple non-portable techniques (such as using SSE registers to hold taint bits) in order to run as fast as possible [9]. Because this system is a full taint analysis tool, its measured overheads *do* include taint propagation and checks. The second baseline system is Umbra, a more general shadow memory system built on top of DynamoRio [49]. Its overheads come from calculating shadow value locations and accessing them on each memory operation. Similar to the tests done by Tiwari *et al.* [40], we test these systems using the SPEC CPU2000 integer benchmarks.

Table 5: Pipelined Events. These are pipelined events that cause overlapping slowdowns. They are logged and their overheads estimated using a trace-based timing simulator.

Event	Symbol
Load watchpoints using SW handler.	$T_{SWcheck}$
Store watchpoints using SW handler.	T_{SWset}
Load watchpoints using hardware	$T_{HWcheck}$
Change page table protection bits.	T_{VM}
Create MLPT entries in MMP.	T_{MLPT}
Write data into bitmaps.	T_{bitmap}

Deterministic Concurrent Execution – We test the deterministic execution tool in a similar manner to Grace by using the benchmark tests included in the Phoenix shared-memory MapReduce suite [33]. We also tested the SPEC OMP2001 benchmarks, another set of programs with the fork-join parallelism that Grace is designed to support.

Demand-Driven Data Race Detection – The default tool we compare against is a commercial data race detector that performs this sharing analysis in software. We test this system in a similar manner to our previous work [19] by running the Phoenix [33] and PARSEC [8] suites.

4.2 Results for Taint Analysis

Figure 6 details the average number of memory operations between each WP cache miss for all of the systems that use some cache mechanism (*i.e.*, not virtual memory). Because these programs have large regions of unwatched data, this tool shows the power of the range cache to cover nearly the entire working set of a program. Similarly, because the MMP PLB can hold aligned ranges, it has a larger reach than MemTracker's cache, which can only hold small sections of per-byte bitmaps. Nonetheless, Figure 7 shows that MMP has worse performance, on average, than MT. This is primarily because some benchmarks cause MMP to spend a great deal of time deciding whether to set or remove upper levels of the trie, in order to best utilize ranges in its PLB.

The taint analysis tool rarely causes our hybrid system to transition from using normal to bitmapped ranges. Because the size of the RC is reduced from 128 to 64 entries, its miss rate is slightly higher than the RC-only system's. The only counterexample is *255.vortex*, which taints a large number of small regions. However, when WLB misses are taken into account (the line "RC+Bitmap" only accounts for range cache misses), the total hit rate drops precipitously. This is because the benchmark has a large number of WP changes, and each change into a bitmapped region causes a WLB eviction.

Figure 7 compares the performance of the HW-assisted watchpoint systems against the two software-based shadow value analysis tools. The lower hit rate in the hybrid system means that it is about 7% slower than one with a larger RC. Nonetheless, the high hit rates of both systems mean that nearly every instruction that is not tainted suffers no slowdown, yielding large speedups over the software analysis tools. It is important to note, however, that MINEMU is also performing taint propagation, overheads which we do not analyze for our hardware systems. Nonetheless, using hardware-supported watchpoints can still result in large performance gains over an always-on tool like Umbra.

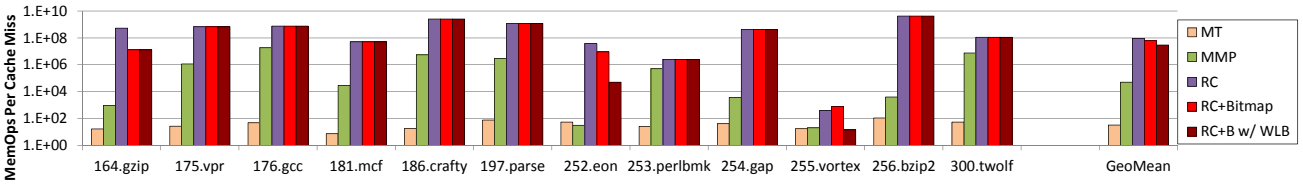


Figure 6: Memory Operations per Watchpoint Cache Miss for Taint Analysis. Most programs store a relatively small number of tainted regions, leading the range cache to have a very high hit rate (some of the benchmarks never miss). "RC+B w/ WLB" counts both range cache misses and the WLB misses, though the latter are filled much more quickly than the former.

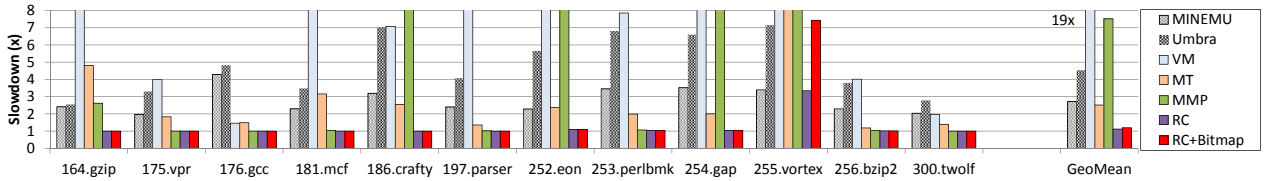


Figure 7: Performance of Demand-Driven Taint Analysis. The patterned are binary instrumentation systems that perform full taint analysis and shadow value lookup, respectively. The maximum slowdowns of the VM system are not shown, as they can go as high as 1400×. The RC + Bitmap solution performs poorly on 255.vortex because each WP change to a bitmapped region causes an eviction from the WLB, and these are quite frequent. Nonetheless, this system easily outperforms manual shadow value lookups.

This performance benefit is dampened in *255.vortex* by its extremely low cache hit rates. All of the demand-driven analysis systems performed poorly for this benchmark, and would probably continue to do so even with high cache hit rates, since the software analysis tool would rarely be disabled – almost 10% of the memory operations in *255.vortex* operate on tainted values. Demand-driven tools are poor at accelerating applications such as this, and would need mechanisms such as sampling to run faster [18].

4.3 Results for Deterministic Execution

Figures 8 and 9 show the cache miss rates and performance, respectively, of the WP-based deterministic execution system. The performance graph is normalized to a Grace-like system that removes watchpoints one page, rather than one cache line, at a time. The normalized performance of the more fine-grained mechanisms will be lower than 1.

Because this system operates on more watchpoints than the taint analysis tool, the cache hit rates are lower. Very few of the watchpoints used in this tool can be held in higher levels of the trie, so MMP's PLB has a much worse hit rate. On average, MemTracker's hit rate is higher than MMP's due to its larger cache. Despite the increased number of watchpoints, the range cache maintains a high hit rate. This can partially be attributed to the tool using watchpoints of 64 bytes in length at minimum, which increases the reach of the range cache in highly fragmented cases.

As Figure 9 illustrates, attempting to use finer-grained watchpoints in the virtual memory system reduces performance significantly (it averages out to 670× slower than using 4KB watchpoints). All of the systems designed for fine-grained watchpoints handle this change much better. The systems that utilize ranges execute at about 90% of the speed of the VM-based system that uses 4KB watchpoints, with the bitmapped range system edging slightly higher. Because this tool primarily works on ranges of data, both range-based systems perform better than the other hardware systems that operate solely on bitmaps.

4.4 Results for Data Race Detection

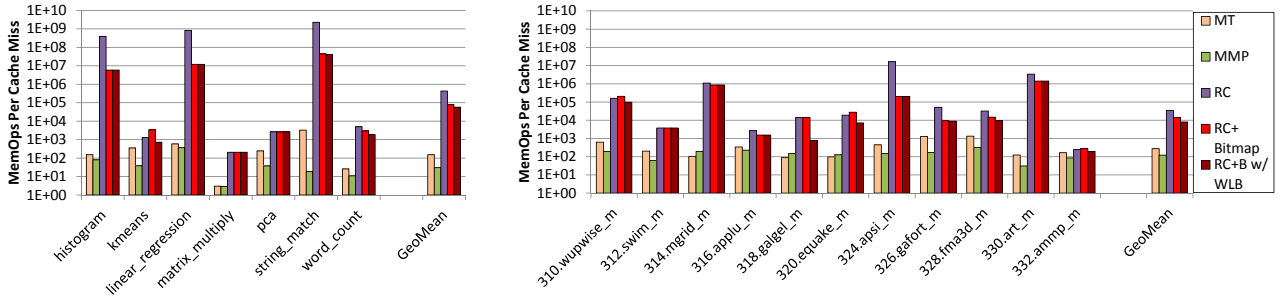
The cache miss rate for a demand-driven data race detector is shown in Figures 10. This tool deals with a much greater number of watchpoints, as it slowly unwatches data touched by individual threads at a byte granularity. The effect this has on the range cache can be easily observed in the Phoenix suite, as it falls from 100,000 memory operations between each range cache miss (in Grace) to 10,000 in this tool. The PARSEC benchmark *canneal* is particularly egregious, as none of the hardware caches go more than an average of 100 memory operations before missing. Because most of these benchmarks have a large number of relatively small ranges, however, this suite shows the benefit of the bitmapped ranges. In PARSEC, the geometric mean of the range cache hit rate is 5× higher when small ranges can be stored as bitmaps, even though the RC itself is half the size.

The performance of the hardware-assisted sharing detectors, shown in Figure 11, is almost always higher than when using software. In particular, the range based systems, even with low hit rates, can still outperform a software system that must check every memory access. The exception in *canneal*, where range-based hardware systems suffer high overheads from the backing store fault handler. In total, however, our hybrid system is able to demonstrate a 9× performance improvement, on average.

4.5 Experimental Takeaways

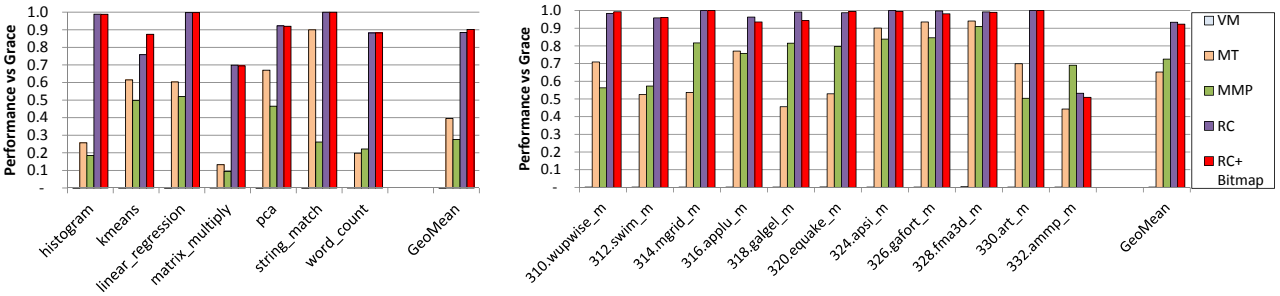
On the whole, the benefits of the range cache are pronounced. Its hit rate is significantly higher than caches that only store WP bitmaps. Perhaps even more importantly, it acts as a write filter. Many of the slowdowns seen by MemTracker are due to writing to the backing store repeatedly. In a similar vein, the algorithm for filling out the trie in MMP can take up a great deal of time when there are many WP changes, and breaking ranges apart can cause a large amount of memory traffic.

Nonetheless, we've found that the addition of a bitmap system to the range cache is beneficial when the watchpoints are small. There are numerous applications that create a large number of small ranges. *dedup* within a race detector, for instance, is significantly helped by the increased reach of the RC when using bitmapped ranges. We found, however, that the WLB miss rate was quite high in many cases. This is partially because writes to bitmapped regions currently evict matching entries from the WLB. It is also the case that



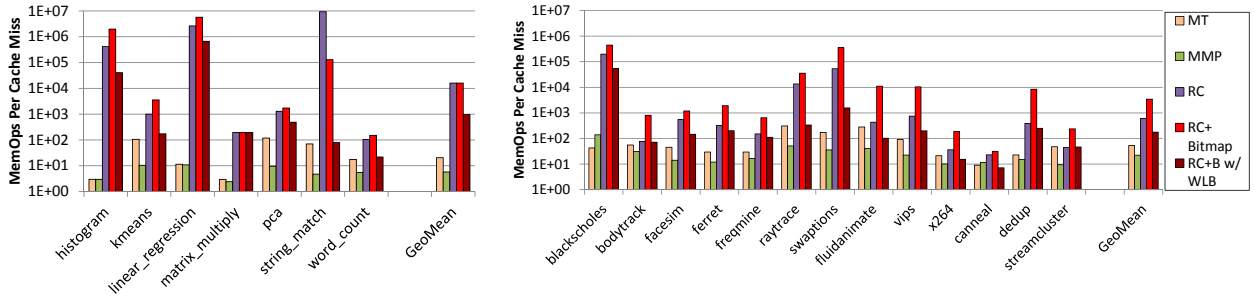
(a) Phoenix Suite **(b)** SPEC OMP2001

Figure 8: Memory Operations per WP Cache Miss for Deterministic Execution. The increased number of WPs causes the RC to have lower hit rates than it had for taint analysis. Because watchpoints are set and removed on cache line sized regions, the addition of a bitmap is only somewhat useful. It does not, however, increase the reach of the system as a whole since the size of the RC is reduced. Nonetheless, both systems take many fewer misses than other watchpoint hardware systems.



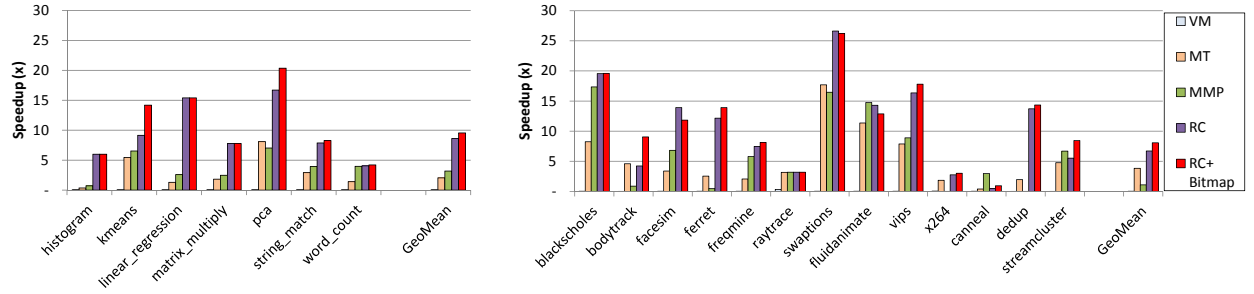
(a) Phoenix Suite **(b)** SPEC OMP2001

Figure 9: Deterministic Execution Performance: Cache Line vs. Page Granularity Watchpoints. The baseline system sets and removes watchpoints at the 4KB page granularity, while these systems operate on 64 byte lines. On average, both range-based systems operate at 90% of the baseline speed while maintaining more accurate checks.s



(a) Phoenix Suite **(b)** PARSEC

Figure 10: Memory Operations per WP Cache Miss for Data Race Detection. This system sets a large number of fine-grained watchpoints, making the bitmap addition especially useful. It increases the number of instructions between each RC miss by over 5× in PARSEC. The total miss rate with WLB included is higher, but those are filled much faster than RC misses.



(a) Phoenix Suite **(b)** PARSEC

Figure 11: Performance of Demand-Driven Data Race Detection vs. Binary Instrumentation. Because the binary instrumentation tool is slow, most of the systems see significant speedups. The WLB miss handler is much faster than RC miss handler, so the bitmapped range system is noticeably faster than the RC system, even though it takes many WLB misses.

that bitmapped ranges are often enabled in sections of code that are difficult to cache. This warrants future research to perhaps find a better way to store these bitmapped ranges.

Though we did not illustrate the tests we did on the OCBM, we found that it was a net benefit, on average. However, its benefit was generally measured in single-digit percentages, rather than the larger gains we often saw with the full bitmap system.

Perhaps the most important area that could be improved in our system is the backing store handler. Range cache misses or overflows required hundreds to thousands of extra cycles to handle (depending on the size of backing store tree and the complexity of the insertion). With overheads this high, the range cache needs a particularly good hit rate to maintain high performance. One of the benefits we found with the bitmapped ranges was that the WLB miss handler was simpler and faster. It is probably possible to build more efficient backing store algorithms, such that they would switch between storage methods depending on the miss rate, eviction rate, and number of watchpoints in the system.

In summary, the hardware-assisted watchpoint systems allowed sizable performance improvements in demand-driven tools. For the deterministic execution system, the finer-grained watchpoints resulted in minor slowdowns, but allowed more accurate sharing analyses than can currently be performed at such a speed. This is the crux of the argument that hardware-assisted watchpoints are a generalized mechanism to improve software tools.

5. Related Works

This section explores works related to watchpoints and their uses. We first discuss hardware proposals that *could* be used to provide unlimited watchpoints, but which have limitations that keep them from being as general as we would like. We then list other applications that could potentially utilize an unlimited watchpoint system.

5.1 Memory Protection Systems

There have been numerous proposals for hardware that provides watchpoint-like mechanisms. Capabilities, for instance, can be used to control software's access to particular regions in memory [15]. Few capability-based systems were built, and even fewer still exist. Most recent publications focus instead on fine-grain memory protection.

One of the most widely cited works in this area is Mondriaan (also spelled Mondrian) Memory Protection [46]. MMP was designed with fine-grained inter-process protection mechanisms in mind, and is optimized for applications that do not perform frequent updates. The protection information is stored in main memory and is cached in a protection lookaside buffer (PLB). MMP utilized a ternary CAM for this cache, allowing naturally aligned ranges to be compactly stored. The first method Witchel proposed for storing protection regions in memory was a sorted segment table (SST), a list sorted by starting address. Though this allows O($ln\ n$) lookups and can efficiently store ranges, it is unsuitable for frequent updates. The second, a multi-level permission table (MLPT), is a trie that holds a bitmap of word-accurate permissions in its lowest level. It is beneficial to use upper levels of this table whenever possible in order to increase the PLB's reach. Checking ranges of permissions in order to "promote" a region can cause significant update slowdowns, however. In general, MMP is designed to work with tools that, while needing memory protection, do not perform frequent updates.

One common method of storing protection data is to put it alongside cache lines [30, 32, 50]. iWatcher, for example, stores per-word watchpoints alongside the cache lines that contain the watched data [50]. These bits are initially set by hardware and are temporarily stored into a victim table on cache evictions. The hardware falls back to virtual memory watchpoints if this table overflows. iWatcher can watch a small number of ranges, which must be pinned in physical memory. If this range hardware overflows, the system falls back to setting a large number of per-word watchpoints. In general, this system is inadequate for tools that require more than a small number of large ranges. Of note, Zhou *et al.* correctly state the usefulness of user-level faults in handling frequently-touched watchpoints. UFO, a similar system, used watchpoints to accelerate speculative software optimization [30]. Unfortunately, because it stores watchpoints in memory by updating the ECC bits of individual lines, it does not allow large ranges to be quickly set or removed. SafeMem uses a similar scheme [32].

MemTracker holds analysis meta-data (which is analogous to watchpoints) in a separate L1 cache, similar to our WLB. It also offers the option of using a hardware state machine to perform meta-data propagation [42]. This means that it can have even less overhead for some types of analyses than a system with a user-level fault handler. However, because its meta-data is stored as a packed array in main memory, it is time-consuming to set or remove watchpoints on large ranges. FlexiTaint, a follow-on work to MemTracker, has the same issues [41].

In order to reduce the number of accesses to the memory protection hardware, Sentry stores its protection data at the cache line granularity [38]. Every line in L1 is unwatched, and the on-chip protection structure is only checked on L1D misses. Their paper gives an excellent analysis of the type of hardware needed to perform watchpoints. However, their system has large overheads when performing frequent updates because it must go to a software handler on every change and evict cache lines accordingly.

5.2 Other Uses for Watchpoints

Watchpoints can also be used to accelerate software systems beyond the three examined in the experiments. Table 3 lists algorithms for the tools discussed in this section.

Hill *et al.* previously made an argument for deconstructed hardware transactional memory (TM) systems [20]. They pushed for HTM additions that could be used for other things, like watchpoints; we instead argue that watchpoints can be used to make (among other things) faster TM systems. A WP-based algorithm to accelerate software TM, as described by Baugh *et al.*, sets WPs on data touched by transactional code; any time another thread touches this data, a conflict will be caught [3]. If transactions are particularly long, forcibly setting watchpoints after every memory access may be slow. In this case, carving working sets out of large watched regions (as we demonstrated for deterministic execution and data race detection) may be faster, as the initial fault times are amortized across multiple memory operations.

Beyond correctness and debugging analyses, speculative parallelization systems allow some pieces of sequential code to run concurrently, and only later check if the result was indeed correct. In essence, watchpoints can be set on values created by the speculative code so that they are verified

before being used elsewhere in the program. Fast Track performs these checks in software using virtual memory watchpoints [23].

Because they allow write faults to be taken on specified read-only regions of memory, watchpoints also enable security systems both as common as bounds checking [14] and as esoteric as kernel rootkit protection [44]. The latter requires little explanation, but it is useful to note that the authors lamented the "protection granularity gap," or the inability to set fine-grained watchpoints.

Watchpoints can even be used, in a fashion, for garbage collection. A semi-space collector will move all reachable objects from one memory space to another at collection time [13, 17]. To implement this efficiently, the program will continue to execute while data is moved, pausing if the program attempts to access data that has not yet been appropriately moved. Appel *et al.* did this by setting virtual memory watchpoints on each memory location that is to be moved [1]. This is similar to the mechanism that Lyu *et al.* use to perform online software updates [26].

6. Conclusion and Future Work

In this paper we presented a hardware design that allows software to utilize a virtually unlimited number of low-overhead watchpoints. By combining the ability of a range cache to store long watched regions with a bitmap's fast access and succinct encoding for small watchpoints, we were able to demonstrate an effective system for a collection of software tools. We used this system to demonstrate that the software community can benefit from generic primitives provided by hardware.

Though our results show that a design of this nature is promising, a number of avenues for future research remain open. Our range cache's software-controlled miss and eviction handler would be unusable without fast fault support. It may therefore be advantageous to look into hardware designs for operating on the backing store. There are also open questions as to the best algorithms for moving from ranges to bitmaps (and vice-versa). We presented a simple algorithm that showed decent results, but it is probably not optimal either in its runtime or its decisions.

Finally, there are potentially promising research results into new types of software tools that could be built on top of watchpoint systems. Software developers and researchers have been valiantly extending the uses of the virtual memory system for decades. This ingenuity may very well yield novel uses for a byte-granularity watchpoint system in the future.

Acknowledgments

We wish to thank Debapriya Chatterjee, Andrea Pellegrini, and the anonymous reviewers, whose suggestions greatly improved this work, as well as Evelyn Duesterwald from IBM Research, for assistance in the final revision. Thanks also to Qin Zhao for providing the Umbra code used in our experiments. The authors acknowledge the support of the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

References

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-time Concurrent Collection on Stock Multiprocessors. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[2] A. W. Appel and K. Li. Virtual Memory Primitives for User Programs. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[3] L. Baugh, N. Neelakantam, and C. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2008.

[4] B. Beander. VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger. In *the Proc. of the Software Engineering Symposium on High-level Debugging*, 1983.

[5] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The Deterministic Execution Hammer: How Well Does it Actually Pound Nails? In *the Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2011.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *the Proc. of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, 53(2):66–75, February 2010.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *the Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[9] E. Bosman, A. Slowinska, and H. Bos. Minemu: The World's Fastest Taint Tracker. In *the Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[10] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, 2009.

[11] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2009.

[12] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *the Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[13] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[14] T. Chiueh. Fast Bounds Checking Using Debug Registers. In *the Proc. of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, 2008.

[15] D. R. Ditzel. Architectural Support for Programming Languages in the X-Tree Processor. In *Spring COMPCON*, 1980.

[16] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-Race Detection for the Kernel. In *the Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[17] R. R. Fenichel and J. C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[18] J. L. Greathouse, C. LeBlanc, T. Austin, and V. Bertacco. Highly Scalable Distributed Dataflow Analysis. In *the Proc.*

of the Symposium on Code Generation and Optimization (CGO), 2011.

[19] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-Driven Software Race Detection using Hardware Performance Counters. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2011.

[20] M. D. Hill, D. Hower, K. E. Moore, M. M. Swift, H. Volos, and D. A. Wood. A Case for Deconstructing Hardware Transactional Memory Systems. Technical report, University of Wisconsin-Madison, 2007.

[21] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection using Demand Emulation. In *the Proc. of the European Conference on Computer Systems (EuroSys)*, 2006.

[22] M. S. Johnson. Some Requirements for Architectural Support of Software Debugging. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1982.

[23] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A Software System for Speculative Program Optimization. In *the Proc. of the Symposium on Code Generation and Optimization (CGO)*, 2009.

[24] E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. In *the Proc. of the USENIX Security Symposium*, 2003.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[26] J. Lyu, Y. Kim, Y. Kim, and I. Lee. A Procedure-Based Dynamic Software Update. In *the Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2001.

[27] R. E. McLear, D. M. Scheibelhut, and E. Tammaru. Guidelines for Creating a Debuggable Processor. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1982.

[28] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *the Proc. of the USENIX Annual Technical Conference*, 1996.

[29] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and Visualizing Full Systems with Data Flow Tomography. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[30] N. Neelakantam and C. Zilles. UFO: A General-Purpose User-Mode Memory Protection Technique for Application Use. Technical report, University of Illinois at Urbana-Champaign, 2007.

[31] R. H. B. Netzer and B. P. Miller. What Are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[32] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Produciton Runs. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[33] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2007.

[34] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[35] E. Schonberg. On-The-Fly Detection of Access Anomalies. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[36] E. Schrock. Watchpoints 101. `http://blogs.oracle.com/eschrock/entry/watchpoints_101`, July 2004.

[37] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-precision. In *the Proc. of the USENIX Annual Technical Conference*, 2005.

[38] A. Shriraman and S. Dwarkadas. Sentry: Light-Weight Auxiliary Memory Access Control. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2010.

[39] B. Sprunt. The Basics of Performance-Monitoring Hardware. *IEEE Micro*, 22(4):64–71, 2002.

[40] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood. A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *the Proc. of the International Symposium on Microarchitecture (MICRO)*, 2008.

[41] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[42] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2007.

[43] R. Wahbe. Efficient Data Breakpoints. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[44] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *the Proc. of the Conference on Computer and Communications Security (CCS)*, 2009.

[45] E. Witchel. Considerations for Mondriaan-like Systems. In *the Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2009.

[46] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[47] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *the Proc. of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[48] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *the Proc. of the ACM/IEEE Conference on Supercomputing*, 1996.

[49] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and Scalable Memory Shadowing. In *the Proc. of the Symposium on Code Generation and Optimization (CGO)*, 2010.

[50] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2004.