Improving the Energy Efficiency of Hardware-Assisted Watchpoint Systems

Vasileios Karakostas[1,2], Sasa Tomic[1], Osman Unsal[1], Mario Nemirovsky[3], Adrian Cristal[1,4]
[1]Barcelona Supercomputing Center
[2]Universitat Politecnica de Catalunya
[3]ICREA Senior Research Professor at Barcelona Supercomputing Center
[4]Spanish National Research Council (IIIA-CSIC)
{first.lastname}@bsc.es

ABSTRACT

Hardware-assisted watchpoint systems enhance the execution of numerous dynamic software techniques, such as memory protection, module isolation, deterministic execution, and data race detection. In this paper, we show that previous hardware proposals may introduce significant energy overheads, and propose WatchPoint Filtering (WPF), a novel filtering mechanism that eliminates unnecessary watchpoint checks. We evaluate WPF on two state-of-the-art proposals for hardware-assisted watchpoints using two common memory checkers. WPF eliminates 83% of the watchpoint checks (up to 99.7%) and reduces 57% of the dynamic energy overhead (up to 78%) on average, without introducing additional performance execution overhead.

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*

General Terms

Design, Performance

Keywords

Watchpoints, Metadata cache, TLB, Filtering, Optimization

1. INTRODUCTION

Writing bug-free code is a difficult task that presumes dedicating a significant amount of time to testing and debugging. However, the demand for higher productivity and for meeting tight release deadlines often results in insufficiently tested software. For instance, a very common type of bugs are memory-related. These bugs often pass development tests and manifest themselves under obscure conditions only after release. To bridge this gap, runtime systems should provide *always-on* and *low-overhead* support for analysis tools that improve the quality of executed code and increase the reliability of the system.

Tools based on dynamic binary instrumentation [1, 5, 6, 8, 12] provide the desired functionality but they impose significant performance degradation (e.g. more than 30× slowdown in Valgrind's MemCheck [12]). The root cause is the instrumentation overhead: all memory accesses are checked in software. Furthermore, commodity processors lack hardware support to accelerate such memory checks. The Translation Lookaside Buffer (TLB) checks access rights at coarse page-level granularity, while debug registers are limited to watching only a few memory locations.

To overcome these limitations, hardware-assisted watchpoint systems [4, 11, 14, 15, 17, 19, 21] have been proposed to enhance analysis tools in production runs. The hardware support typically includes a metadata-cache that accelerates watchpoint checks. Although these proposals seem to have good performance, they often make unneeded watchpoint checks and introduce significant dynamic energy overhead. We find that a high performance scheme that accesses the metadata-cache in *every* memory operation introduces up to 15.8% dynamic energy overhead - only due to the metadata-cache - with respect to the dynamic energy consumed in the private per-core caches. Taking into account the increasing importance of energy consumption in future processors, vendors are expected to employ hardware support for unlimited watchpoints only if they can provide both minimal energy and performance execution overheads.

In this paper, we propose WatchPoint Filtering (WPF), a novel mechanism that filters unnecessary watchpoint checks. Our proposal leverages the existing hardware in most commodity processors – the Virtual Memory mechanism. WPF repurposes an unused bit in the TLB entry, to mark whether the page has any defined watchpoints. This way, WPF manages to eliminate unnecessary watchpoint checks, reduces the number of accesses to the metadata-cache and improves the energy efficiency and performance of the system. Being orthogonal to the previous proposals for watchpoint support, WPF can be applied to almost any proposed implementation.

The main contributions of this paper are:

- We identify a gap between energy efficiency and performance in the state-of-the-art hardware-assisted watchpoint systems, and propose a filtering mechanism that eliminates unnecessary watchpoint checks.
- We demonstrate how WPF integrates with two previously proposed hardware watchpoint mechanisms and show that applying WPF requires only minor modifications to the original proposals.

- We evaluate the performance of WPF with two memory checkers. WPF eliminates on average 83% of the metadata checks and reduces 57% of the dynamic energy overhead, without introducing additional performance overhead.

In Section 2 we discuss the shortcomings of the hardware-assisted watchpoint systems. We describe the WPF mechanism and demonstrate how WPF can be applied to two state-of-the-art proposals in Section 3. We evaluate the efficiency of WPF in Section 4. In Section 5 we discuss related work and, finally, in Section 6 we conclude our study.

2. BACKGROUND & MOTIVATION

In this section we provide information about watchpoint systems, and we elaborate on the motivation for our work.

2.1 Watchpoints

Watchpoints allow the detection or the prevention of accesses to certain memory locations in user-level [4, 11, 15, 17, 21]. A memory watchpoint is defined by: (i) a memory range that needs to be watched, (ii) the desired access rights, and (iii) an exception handler. An access violation invokes a lightweight user-level handler. This handler may report an error, start a debugger, or perform a healing action to automatically recover from the access violation. All watchpoint information (the metadata) is stored in data structures managed by the software runtime.

Watchpoints can enhance a multitude of analysis tools to operate efficiently. For example, Venkataramani *et al.* [17] explain how watchpoints can prevent common memory bugs from manifesting. Shriraman *et al.* [15] apply watchpoints to enhance multi-module software engineering in Apache. Greathouse *et al.* [4] show how watchpoints can accelerate dynamic dataflow analysis, deterministic execution, and data race detection tools among others.

To accelerate the *always-on* execution of such analysis tools, watchpoint systems employ additional hardware support that mitigates the overheads of checking memory accesses in software. The hardware support typically includes a *metadata-cache* [4, 15, 17, 19, 21]. In case the hardware detects a watchpoint violation or a metadata-cache miss, the software runtime takes control of the execution.

2.2 Motivation

There are two main categories of hardware-assisted watchpoint systems. The first category targets high-performance; the processor accesses the metadata-cache in parallel with the L1 cache on *every* memory access [4, 17, 19, 21]. MemTracker [17] is a typical representative of this category. We find that a MemTracker-like approach introduces up to 15.8% of dynamic energy overhead only due to the metadata-cache with respect to the dynamic energy consumed in the private per-core caches (i.e. TLB, L1 and L2 cache) during normal execution (Section 4.2).

The second category targets energy efficiency; a representative example of this approach is Sentry [15]. The metadata-cache is consulted in the L1 miss path trading off performance for energy reduction. Indeed, the Sentry-like approach increases performance overhead around 2× compared to MemTracker [15]. Furthermore, we find that in such an approach, the total dynamic energy overhead can be up to 13.6% due to the high interaction of the metadata-cache with the L2 cache (Section 4.2).

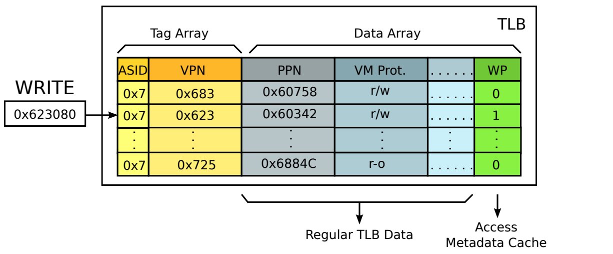


Figure 1: TLB filtering of memory-watchpoint checks. The WP-bit marks whether there are watchpoints in the corresponding page. The WP-bit is set and cleared by user-level instructions.

Therefore, we identify a gap between energy efficiency and high performance in the proposed hardware-assisted watchpoint systems. In this paper we narrow the gap with a filtering mechanism that orthogonally improves the energy efficiency in both categories without affecting the performance.

3. WATCHPOINT FILTERING

In this section we describe the WatchPoint Filtering (WPF) mechanism, we demonstrate how WPF integrates into two state-of-the-art hardware-assisted watchpoint systems, and we explain the implementation details of WPF.

3.1 The Idea

WPF reuses the Translation Lookaside Buffer (TLB) and assigns novel functionality to a currently unused (reserved) bit in the TLB entry. We name it as WatchPoint (WP) bit and it indicates the existence of watchpoints in the corresponding page (Figure 1). In case the WP-bit is set, the page contains defined watchpoints that should be checked in the metadata-cache. Otherwise, there are no watchpoints set and no further checks are performed.

WPF improves the energy efficiency in several ways. First, WPF filters out metadata checks for pages that do not have defined watchpoints. This reduces the dynamic energy spent in the metadata-cache. Second, WPF eliminates the pollution of the metadata-cache with non-restrictive watchpoints, i.e. useless entries that do not enforce any restriction in memory accesses. This results in higher metadata hit-ratio and enables the employment of smaller metadata-caches without affecting the performance. Third, by improving the metadata hit-ratio, WPF also filters out the transitions to the software runtime, reducing further the execution overhead and the cache interference due to these routines. Finally, by reusing the existing hardware of TLB (extending it with simple logic), WPF increases neither the latency, nor the static power dissipation, or the area of the TLB.

WPF avoids unnecessary checks in the metadata-cache leveraging the observation that watchpoints are typically set on a limited part of the address space. The amount of the reduced checks depends on how the analysis tool utilizes the watchpoints. Since WPF is an unintrusive optimization mechanism, the resulting system achieves lower dynamic power dissipation and better overall efficiency. Being orthogonal to previous mechanisms, WPF can be applied to almost any existing watchpoint implementation.

3.2 Integrating WPF

WPF with MemTracker-like approach. MemTracker [11] is the main representative of high performance hardware-assisted watchpoint designs. In MemTracker, the metadata-

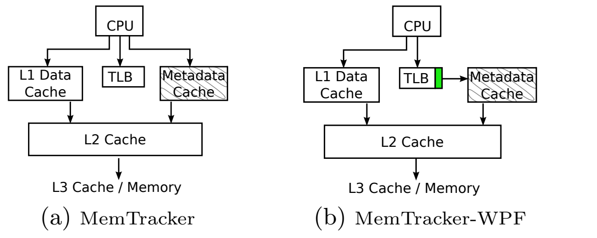


Figure 2: MemTracker architecture with and without WPF. The hardware support for watchpoints is hatched, while WPF is shaded in gray.

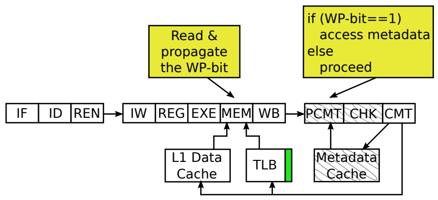

Read & propagate the WP-bit
if (WP-bit==1) access metadata else proceed
IF
ID
REN
IW
REG
EXE
MEM
WB
PCMT
CHK
CMT
L1 Data Cache
TLB
Metadata Cache

Figure 3: Integration of WPF into the processor pipeline of MemTracker. The processor fetches the WP-bit in the MEM stage of the pipeline when the TLB is accessed. In the pre-commit stage (PCMT), the WP-bit is checked. According to its value, the metadata-cache is accessed. Finally, the memory access is checked in the CHK stage.

cache is accessed in parallel with the L1 cache on every memory operation (Figure 2a). Although the metadata-cache typically has a small size (around 4KB) it can introduce a significant amount of dynamic energy consumption. In Section 4 we show that such an approach introduces up to 15.8% overhead of dynamic energy consumption with respect to the per-core cache structures (i.e. TLB, L1 and L2 cache).

In MemTracker with WPF, the watchpoint checks take place only if the WP-bit is set (Figure 2b). MemTracker checks for watchpoint violations in the last stage of the execution pipeline, just before an instruction commits. To integrate WPF with a MemTracker-like system, no additional pipeline stages are required (Figure 3). In this way, WPF reduces the number of accesses to the metadata-cache and improves the energy efficiency of MemTracker.

WPF with Sentry-like approach. Sentry [15] targets energy efficiency and caches watchpoints in a metadata-cache placed on the L1 miss path. The metadata-cache reuses the L1 cache coherence states to elide checks on L1 hits. For example, if a cache-line is in shared state, it can be read directly without consulting the metadata-cache. If the watchpoint access rights are down-graded, the corresponding L1 cache lines are invalidated. The next memory reference to this cache line causes a miss in L1 and a check in the metadata-cache.

Figure 4 shows the design of Sentry and its enhanced version with WPF. Although Sentry is able to reduce the metadata-cache accesses compared to a performance-aggressive design (such as MemTracker), WPF can improve the dynamic energy and performance overheads even further.

Discussion. When we refer to MemTracker and Sentry in this paper, we focus on when the metadata-cache is accessed: (i) in parallel with the L1 cache (MemTracker), or (ii) in the L1 miss path (Sentry). The rest of their key mechanisms (e.g. the programmable state machine in MemTracker or the protection domain support in Sentry) are independent of our proposal. However, in Section 5 we explain how WPF

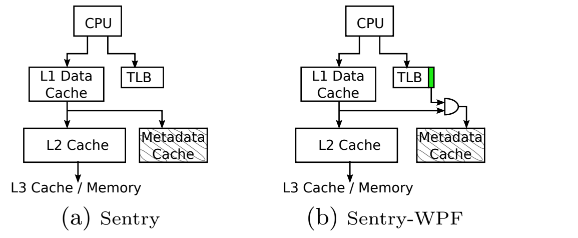


Figure 4: Sentry architecture with and without WPF. The metadata-cache is accessed only when a memory reference misses in L1 and the WP-bit for the corresponding page is set.

compares to their optimization techniques for reducing the number of accesses to the metadata-cache.

3.3 Implementation Details

WP-bit in memory. The software runtime holds various data structures that associate watchpoints with access rights and exception handlers. Thus, the software runtime is slightly extended to hold and control the WP-bit information (one bit per virtual page). In case of a TLB miss, the WP-bit is set and the software runtime is invoked to update it lazily without affecting the TLB-miss critical path.

Updating the WP-bit. Every time the programmer modifies a watchpoint, the software runtime serves the requested modification by updating its data structures and the metadata-cache. This is when the software runtime updates the corresponding WP-bit through a new unprivileged instruction added to the Instruction Set Architecture, which modifies the WP-bit from the user-level code. Security issues are prevented using the same mechanisms that disable arbitrary code to update watchpoints (e.g. state permission in MemTracker, or protection domains in Sentry).

Multi-core configurations. In multi-core systems, each processor has its own private TLB. To maintain the WP-bit coherent across the TLBs, we should invalidate the relevant entries. This could happen using the classic TLB shootdown approach, where the initiator core uses inter-processor interrupts to notify the rest of the cores to update the WP-bit of their entry. Note that the shootdown algorithm takes place immediately only when the check of the watchpoints is enforced (WP-bit goes from 0 to 1) so that no watchpoint checks are missed. The best approach is to build WPF on top of hardware-coherent TLBs, as proposed in [13] and [18]. This allows better performance execution by eliminating the need for inter-processor interrupts.

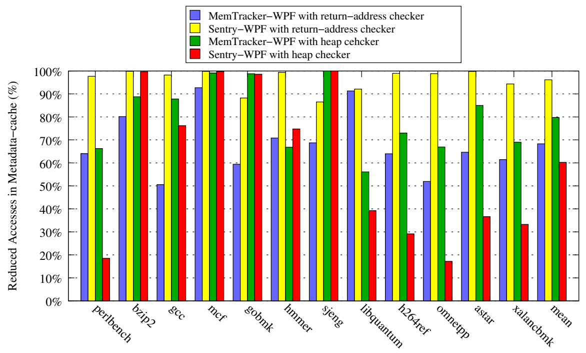
Large pages. Modern operating systems provide support for large pages [2, 10]. The use of large pages somewhat reduces the benefits of WPF, since the WP-bit represents a larger memory range. However, we show that WPF provides decent advantages even with large pages in Section 4.2.

4. EVALUATION

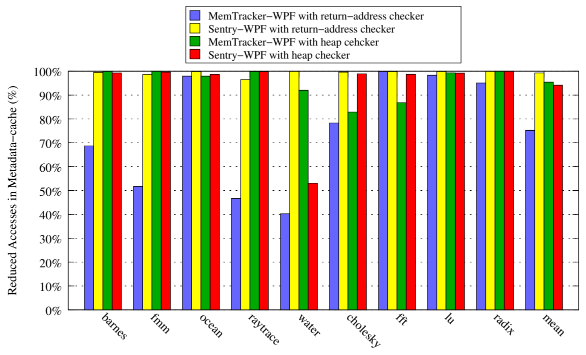
In this section we evaluate how WPF improves the dynamic energy consumption and the performance of hardware-assisted watchpoint systems.

4.1 Simulation Methodology

Watchpoint Tools and Benchmarks. We implement two memory checkers. The first is the *return-address checker*; it protects the control flow of an application by disallowing writes to the return addresses of functions (stored in



(a) SPECint2006



(b) SPLASH2

Figure 5: Percentage of reduced checks in the metadata-cache due to WPF.

Feature	Description
Processor(s)	8 cores, x86 in-order
L1 Cache	32 KB, 64B cache line, 4-way assoc., private, writeback, 2 cycles latency
L2 Cache	256 KB, 64B cache line, 8-way assoc., private, writeback, 8 cycles latency
L3 Cache	8 MB, 64B cache line, 16-way assoc., shared, writeback, 16 cycles latency
Data TLB	256 entries, 4-way assoc., 4 KB page, accessed in parallel with L1
main memory	4GB, 200 cycles latency
MemTracker [17]	4KB, 4-way assoc., in parallel with L1
Sentry [15]	4KB, 4-way assoc., on L1 miss path

Table 1: Simulator Configuration.

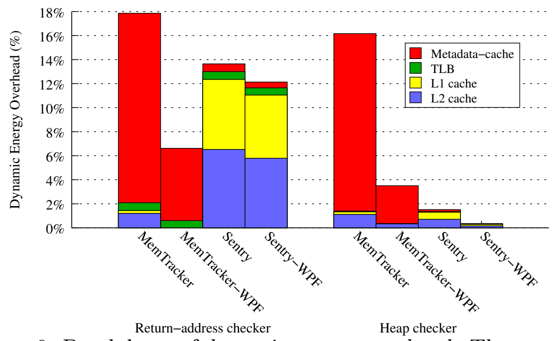


Figure 6: Breakdown of dynamic energy overhead. The overhead is estimated with respect to the per-core caches (i.e. TLB, L1 and L2 cache). WPF reduces significantly the dynamic energy consumption in both the metadata-cache and the cache hierarchy.

the stack) with watchpoints. The second is the *heap checker* that sets a no-access watchpoint in each memory deallocation to prevent dangling pointer usage. In a memory allocation, the checker removes possible watchpoints so that the allocated memory can be used, and prevents from out-of-bounds accesses by setting no-access watchpoints before and after the allocated area. On top of these memory checkers, we run the SPECint2006 benchmarks [7] with the test inputs and the Splash2 [20] benchmarks with the default inputs.

Simulation Infrastructure. To evaluate the efficiency of WPF, we use Pin [8] to implement the two checkers (by capturing the events that trigger watchpoint updates according to the checker) and to simulate an x86 chip multiprocessor. The simulated system consists of simple x86 in-order cores with IPC=1 except on memory accesses. The TLB and the data cache hierarchy are modeled in detail. Caches are inclusive and are kept coherent through a MESI directory-based protocol (Table 1). To evaluate the energy savings of WPF, we use CACTI 6.5 [9] with 32nm technology. For TLB, L1 caches, L2 caches, and the MemTracker's metadata-cache we use high-performance transistors ("itrs-hp"), while for the metadata-cache of Sentry we use low-power transistor technology ("itrs-lop").

4.2 Evaluation Results

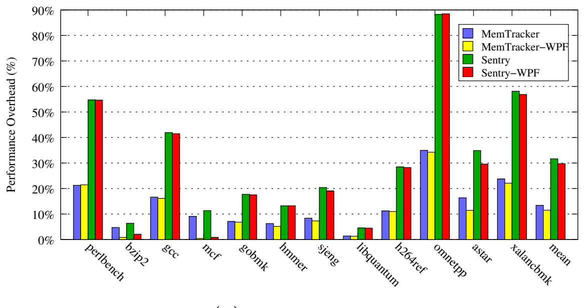
We first evaluate how efficiently WPF filters unnecessary checks in the metadata-cache, and what improvements WPF provides in dynamic energy consumption. We then show how WPF affects the system's performance and increases the hit-ratio of the metadata-cache. Finally, we make a sensitivity analysis of the parameters that WPF depends on.

Reducing Metadata Checks. Figure 5 shows the percentage of reduced watchpoint checks that WPF achieves for MemTracker and Sentry, using the return-address checker and the heap checker.

WPF eliminates 79% of metadata checks for MemTracker and 87% for Sentry on average. The efficiency of WPF with the evaluated checkers depends on various characteristics of the applications, such as the percentage of stack/heap accesses, the frequency of calling functions and the frequency of memory allocations/deallocations. For example, MemTracker-WPF with the return-address checker shows less benefits for applications that call functions frequently, such as `gcc`, `omnetpp` and `xalancbmk`. However, even for these applications, WPF eliminates more than 50% of the metadata checks. Similarly, Sentry-WPF with the heap checker exhibits smaller improvements for the applications that stress the memory allocator, but still eliminates more than 61% of metadata checks on average for this checker.

Reducing Dynamic Energy. Figure 6 shows the dynamic energy overhead spent in the per-core private caches (i.e. TLB, L1, L2 and metadata-cache) with respect to that of the normal execution without watchpoint support and checkers. The figure shows important findings for both original schemes and the potential benefits of WPF.

Regarding the original schemes, we find that MemTracker introduces up to 15.8% of dynamic energy overhead spent only in the metadata-cache. In Sentry the energy overhead of the metadata-cache is significantly lower than in MemTracker, but the total dynamic energy overhead can be up to 13.6% due to the high interaction with the L2 cache. Hence, we conclude that the hardware-assisted watchpoint systems may introduce significant energy overheads.


(a) SPECint2006

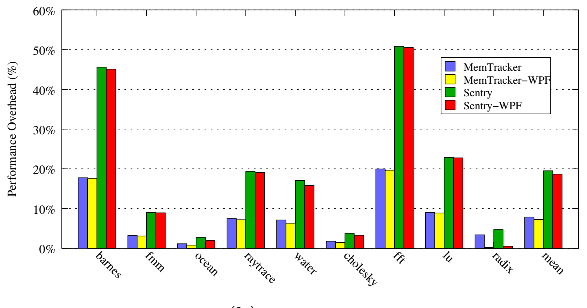

(b) SPLASH2

Figure 7: Performance overhead for the return-address checker. The overheads for the heap checker are less than 2% on average across all configurations. WPF achieves slight performance improvement by reducing the pressure in the metadata-cache.

WPF reduces by 71% and by 44% the total dynamic energy overhead of MemTracker and Sentry respectively. While WPF reduces the dynamic energy spent in the metadata checks, it cannot eliminate the metadata updates. However, the checks are more frequent than the updates, and therefore WPF brings significant improvements in the dynamic energy overhead of the metadata-cache itself. Besides this, WPF also achieves to reduce the watchpoint-induced overheads in the rest of the memory hierarchy by improving the hit-ratio of the metadata-cache. Better hit-ratio translates to less memory accesses and less interference with the application's data in the cache hierarchy because less metadata misses have to be resolved.

Figure 6 shows also that MemTracker-WPF can be more energy efficient than Sentry-WPF (6.6% vs. 13.7%) with tools that update watchpoints frequently (return address checker), and less energy efficient (3.5% vs. 0.4%) with tools that update watchpoints less often (heap checker). This tradeoff should be considered in the implementation of hardware-assisted watchpoint systems.

Improving Performance. Among the two evaluated checkers, the return-address checker demands more frequent watchpoint updates and introduces higher overheads. Figure 7 compares the performance overhead of MemTracker and Sentry with and without the WPF mechanism running the return-address checker.

The results show that WPF achieves a slight performance improvement for MemTracker and Sentry. The main sources of overhead for the original schemes are: (i) the software runtime updates, and (ii) the implications of accessing the metadata-cache in every memory operation (in every L1 miss for Sentry), i.e. increased pressure in the metadata-cache, unnecessary resolutions of metadata-cache misses and interference with the data in the memory hierarchy. WPF can reduce only the second source of overhead. On average, WPF reduces the performance overhead from 6.1% to 5% for MemTracker, and from 13.6% to 12.5% for Sentry.

Increasing Metadata Hit-Ratio. WPF eliminates the pollution of the metadata-cache with non restrictive watchpoint entries by not accessing the metadata-cache on every memory operation. Figure 8 shows the hit-ratio of the metadata-cache with various sizes, from 4KB (default) to 256B. The results show that WPF increases the hit-ratio of the metadata-cache for each size configuration. Moreover, WPF maintains high hit-ratio as the metadata-cache size reduces - for MemTracker the hit-ratio remains practically the same. This way, WPF can further improve the energy efficiency of the watchpoint system by employing a smaller metadata-cache, while maintaining similar performance.

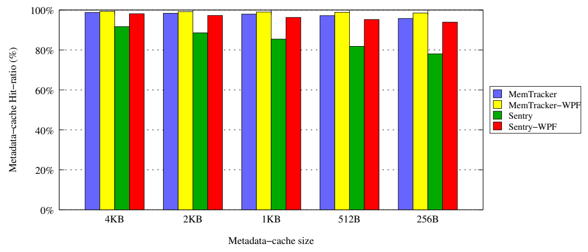


Figure 8: Sensitivity analysis for the metadata-cache hit-ratio. WPF can further improve the energy efficiency by employing a smaller metadata-cache, while maintaining similar performance.

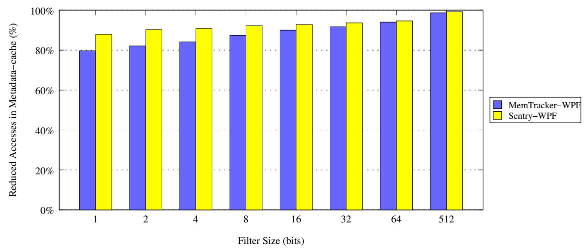


Figure 9: Sensitivity analysis for the filter size. The proposed WPF mechanism with 1 bit per TLB entry is sufficient regarding the tradeoff of performance against area and power.

Sensitivity Analysis for Filter Size. The efficiency of WPF comes from its ability to filter out unnecessary metadata checks by repurposing a single unused bit in each TLB entry. We evaluate WPF with various sizes of filters, where the filter applies as a mask to the corresponding TLB entry.

Figure 9 shows the results of the sensitivity analysis for the filter size ranging from 1 to 512 bits which represents an ideal per-word filter. As the filter size increases, the percentage of reduced accesses to the metadata-cache also increases. We found no significant differences for 1 to 8-bits filters, however for filters that are larger than 16-bits, the improvement becomes significant, while an ideal filter achieves more than 98% reduced metadata accesses. Considering the tradeoff of performance against area and power, we conclude that WPF with 1 bit per TLB entry is sufficient.

Sensitivity Analysis for Page size. WPF's improvements also depend on the page size. Figure 10 shows the results for various page sizes from 4KB to 4MB. Regarding the return-address checker, WPF is not significantly affected by the page size since most of the watchpoints are concentrated in a small region of the memory space. Even with 4MB page size, WPF reduces more than 66% of the metadata checks for MemTracker and 93% for Sentry. On the other hand, the

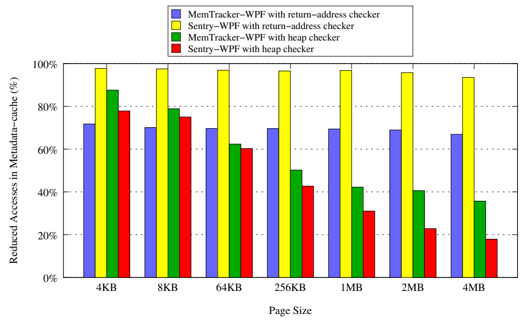


Figure 10: Sensitivity analysis for page size. Even with large pages, WPF eliminates a significant amount of metadata checks.

efficiency of WPF with the heap checker reduces as the page size increases. However, WPF still eliminates the metadata checks by 36% for MemTracker and 18% for Sentry.

5. RELATED WORK

Several proposals reduce metadata checks using dedicated registers. iWatcher [21] introduces a register to completely turn on/off the watchpoint system. Mondrian [19] uses sidecar registers (SRs) to cache the metadata for an arbitrary range of memory addresses. MemTracker [17] can execute multiple checkers concurrently. To minimize overheads, MemTracker uses an event mask register that masks out unused checkers. WPF acts complementary to the event mask register without completely turning off a checker.

HeapMon [14] adds a bit per cache line to indicate that a line access should be checked in software. Chen *et al.* proposed idempotent filters (IF) to eliminate redundant metadata checks in Log-Based Architectures [3]. WPF acts complementary to IFs for deciding whether the metadata check that skipped IF need to be performed or not.

SoftSig [16] relies on the programmer to mark code regions whose memory accesses should be checked in order to reduce the accesses to the signature file. WPF depends on the distribution of watchpoints across the address space. Thus, WPF improves SoftSig in an orthogonal way.

Greathouse *et al.* [4] accelerate watchpoint support introducing two different on-chip caches: a bitmap and a range cache. Incorporating WPF in [4] is straightforward, since the range cache contains the necessary information for updating the WP-bit.

Sentry [15] utilizes the F-bit that has some relevance to the WP-bit in the sense that they both reduce accesses to the metadata-cache. However, the F-bit is stored in the Page Table Entry (PTE). Updating the F-bit requires a slow call to the *operating system*; the authors claim that an optimized low-overhead system call takes around 300 cycles in modern microarchitectures. In addition, they do not further evaluate the benefits of the F-bit. In this paper, we propose and quantify the benefits of having *lightweight user-space* control of the WP-bit, we explore the WPF mechanism as a universal optimization method applicable to a general hardware watchpoint mechanism, and we show that WPF enables high-performance hardware-assisted watchpoint systems being more energy efficient than those that target solely energy efficiency under tools that update watchpoints frequently.

6. CONCLUSIONS

In this paper we presented WPF, a filtering mechanism for hardware-assisted watchpoint systems. Using only one bit per TLB entry, we showed that WPF eliminates up to 99.7% of the watchpoint checks, and reduces up to 78% of the dynamic energy overhead. WPF introduces nearly no area or static power dissipation overhead and can orthogonally enhance almost any existing watchpoint implementation.

7. ACKNOWLEDGMENTS

The authors would like to thank Srdjan Stipic, Nehir Sonmez, Adria Armejach and Daniel Nemirovsky for their feedback. This work was partially supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2007-60625 and TIN2008-02055-E, and by the European Network of Excellence on High Performance Embedded Architecture and Compilation (HiPEAC).

8. REFERENCES

[1] D. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, MIT, 2004.
[2] K. Chen et al. Improving enterprise database performance on intel itanium architecture. In *Ottawa Linux Symposium*, 2003.
[3] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, et al. Flexible hardware acceleration for instruction-grain program monitoring. ISCA, 2008.
[4] J.L. Greathouse, H. Xin, Y. Luo, and T. Austin. A case for unlimited watchpoints. ASPLOS, 2012.
[5] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. ICSE, 2002.
[6] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. USENIX, 1991.
[7] J. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, Sept. 2006.
[8] C. Luk, R. Cohn, R. Muth, H. Patil, et al. Pin: building customized program analysis tools with dynamic instrumentation. PLDI, 2005.
[9] N. Muralimanohar et al. Architecting efficient interconnects for large caches with cacti 6.0. *IEEE Micro*, 28:69–79, January 2008.
[10] J. Navarro et al. Practical, transparent operating system support for superpages. OSDI, 2002.
[11] N. Neelakantam and C. Zilles. Ufo: A general-purpose user-mode memory protection technique for application use. Technical report, UIUC, 2007.
[12] N. Nethercote et al. Valgrind: a framework for heavyweight dynamic binary instrumentation. PLDI, 2007.
[13] B. Romanescu, A. Lebeck, D. Sorin, and A. Bracy. Unified instruction/translation/data (unitd) coherence: One protocol to rule them all. HPCA, 2010.
[14] R. Shetty et al. Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *Ibm Journal of Research and Development*, 50:261–276, 2006.
[15] A. Shriraman and S. Dwarkadas. Sentry: light-weight auxiliary memory access control. ISCA, 2010.
[16] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. Softsig: software-exposed hardware signatures for code analysis and optimization. ASPLOS, 2008.
[17] G. Venkataramani et al. Memtracker: Efficient and programmable support for memory access monitoring and debugging. HPCA, 2007.
[18] C. Villavieja et al. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. PACT, 2011.
[19] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. *Sigplan Notices*, 37:304–316, 2002.
[20] S. Woo et al. The splash-2 programs: characterization and methodological considerations. ISCA, 1995.
[21] P. Zhou, F. Qin, et al. iwatcher: Efficient architectural support for software debugging. ISCA, 2004.