Exploring the Performance Implications of Memory Safety Primitives in Many-core Processors Executing Multi-threaded Workloads

Masab Ahmad
University of Connecticut
masab.ahmad@uconn.edu

Syed Kamran Haider
University of Connecticut
syed.haider@uconn.edu

Farrukh Hijaz
University of Connecticut
farrukh.hijaz@uconn.edu

Marten van Dijk
University of Connecticut
vandijk@engr.uconn.edu

Omer Khan
University of Connecticut
khan@uconn.edu

ABSTRACT

Security is a vital consideration for today's processor architectures, both at the software and hardware layers. However, security schemes are known to incur significant performance overheads. For example, buffer overflow protection schemes perform software checks for bounds on program data structures, and incur performance overheads that are up to several orders of magnitude. To mitigate these overheads, prior works focus on either changing the security scheme itself, or selectively apply the security scheme to minimize program vulnerabilities. Most of these works also focus primarily on single core processors, with no prior work done in the context of multicore processors. In this paper, we show how increasing thread counts can help hide the latency overheads of security schemes. We also analyze the architectural implications in the context of multucores, and the insights and challenges associated with applying these security schemes on mutithreaded workloads.

1. INTRODUCTION

Security requirements are now pervasive in today's computing infrastructure. Computer architects are integrating ever more complex security primitives in the software and hardware layers [2]. With the advent of multicore processors and other integrated shared resources, such potential susceptibilities increase. These vulnerabilities allow malicious entities to tamper with processors in many ways, such as memory tampering [24], buffer overflows [25], and side channel leakage exploits [6]. However, in this paper we only discuss program vulnerabilities in C/C++ codes, since application and legacy codes present the hardest security challenges. Security schemes have been developed to detect and prevent attackers from gaining access to private data [23]. These schemes include dynamic information flow tracking [20, 28], context based detection [24], pointer protection [12], and stack protection [7], to name a few.

HASP '15, June 13, 2015, Portland, OR, USA

ACM 978-1-4503-3483-9/15/06..$15.00
http://dx.doi.org/10.1145/2768566.2768572.

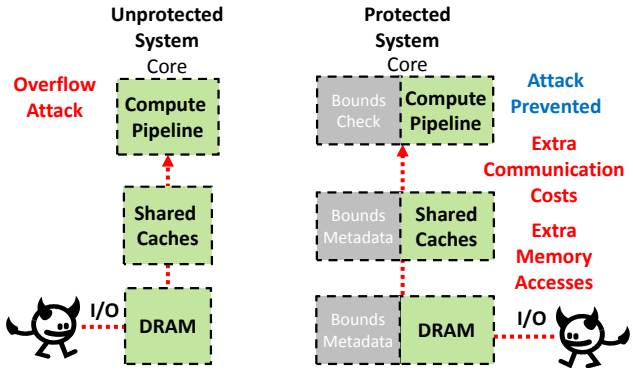

Figure 1: Bound checking prevents the buffer overflow attack. However, it incurs performance overheads associated with managing the security metadata and checks.

Program vulnerabilities in C/C++ codes mostly arise from spatial safety compromises [31]. This leads to program crashes, incorrect results, and memory corruption. Inside programs, spatial safety exploits arise from out-of-bound data structure accesses, dangling pointer dereferences, and access violations. Information flow tracking schemes, such as MemTracker [28], taint suspicious data from the I/O, and raise an exception when tainted data propagates into program control flow. However, information flow tracking schemes suffer from high false positive detection rates, raise expensive exceptions, and thus are not used in everyday applications. Bounds checking methods, such as Baggy Bounds Checking [1], deal with security issues by inserting array bound and pointer access checks at compile time. Their ease of use and near-zero false positive rates precede their use in many applications [16]. The most notable of bound checking memory safety schemes is SoftBound, an open source software catering for both C and C++ programming languages [17, 5]. SoftBound optimizes data structure bounds checking by considering spatio-temporal behaviors in memory accesses. It's resulting performance overheads, using sequential SPEC2006 and Olden [21] benchmarks averages around 32%. Since SoftBound is highly optimized with minimal overheads, we use it as the memory safety scheme in this paper.

With the proliferation of multicore processors, it is imperative that security schemes scale seamlessly on such architectures. Previous research has mainly focused on reducing performance over-

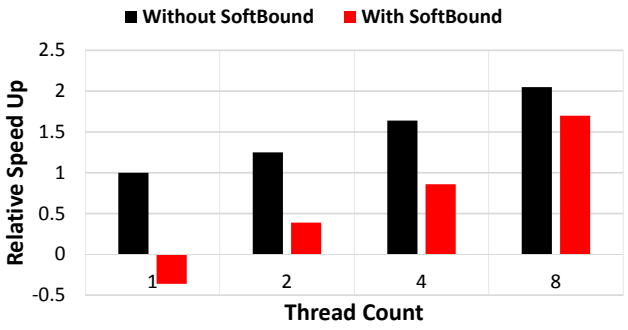

Figure 2: Exploiting multicore concurrency for SoftBound protected parallel Prefix Scan benchmark.

heads of security schemes in sequential benchmarks, and lack insights about the challenges multicore implementations face. As shown in Fig. 1, these challenges include data access and communication overheads due to the shared hardware resources, resulting in performance loss for parallel programs. With malicious exploits increasing at a rapid pace, such applications must utilize security schemes with minimal performance interference. Working with parallel workloads also offer several opportunities to hide latencies generated by these security primitives. For example, multicore concurrency can be used to hide the latency overheads of the security schemes.

To motivate our concurrency argument, we take the example of a parallel implementation of Prefix Scan benchmark, executing on an 8-core Intel machine. Prefix Scan is a widely used benchmark and has scalable parallel implementations, and hence we use it as a motivating example with SoftBound. Fig. 2 shows the speedup when the benchmark is compiled with and without SoftBound. The speedup is measured relative to a sequential version compiled without SoftBound. Fig. 2 shows that with 8 threads the speedup with SoftBound is almost equal to the speedup at a 4 threaded version without the SoftBound overheads. The difference between the normal and SoftBound versions is also decreasing, and it is expected that the completion time of the SoftBound version will equal or become better than the normal version. This shows how increasing thread concurrency can hide the latencies of buffer overflow protection schemes. The Prefix Scan benchmark shown is highly parallel, and we show how other benchmarks, some of which are not scalable, can also benefit from this insight.

Contributions: We propose to use concurrency as an opportunity to possibly hide the latency of program protection schemes and use several benchmarks comprising of commonly used parallel programs to evaluate our insights. We show that well-known benchmark suites, such as PARSEC [3] and SPLASH-2 [29] have vulnerabilities that lead to out-of-bound memory errors, and need to be addressed. Evaluations and characterizations are done for both simulator and real machine based setups. We also show that if the benchmark memory footprint is significantly large, then improving performance through multithreading becomes a challenge. Programs with large working sets have more security checks inserted, which degrades performance. Performance impacts become even more pronounced if the critical sections of the parallel program, such as atomically locked code regions, require security checks.

We motivate two possible solutions to address these challenges. First, the data structures needed to support security checking must be organized such that it mitigates the rate with which the off-chip memory is accessed. Second, the processor must guarantee enough bandwidth to efficiently access the security metadata. For example, a dedicated hardware prefetcher that precisely predicts the access patterns for security metadata can significantly improve performance. Moreover, if the performance of operations on the security metadata is improved, the critical section performance also improves and less synchronization overheads are incurred.

Paper Organization: Section 2 describes several bounds checking schemes and our reasons for choosing SoftBound. Section 3 provides our methodology and evaluation setup. Section 4 shows and characterizes several benchmarks we use to evaluate our arguments. Future Work is described in Section 5, and Section 6 concludes the paper.

2. RELATED WORK

Significant research has been done on software-level memory safety schemes using capability systems. Among the various proposals are software only approaches [16, 17, 1, 18, 19, 22, 4], and partial or full hardware-assisted approaches [5, 30, 15]. Software approaches include both compiler and application based schemes, while hardware approaches tweak the underlying hardware to ensure secure protection of running applications. Hardware approaches, such as CHERI [30] and HardBound [5], result in much lower performance overheads. However, these specialized schemes have high area overheads, require substantial modifications to the operating system and Instruction Set Architecture (ISA), and also modify the processor hardware significantly. The rest of these approaches each strike different balance of protection, performance, and compatibility with existing applications. A previous survey by Szekeres *et al*. [26] shows the importance of bound checking schemes, analyzes the current state of protection mechanisms, and the corresponding exploit techniques that have been used to defeat them. It identifies pointer-based checking as the only class of approaches to provide complete, non probabilistic detection of both spatial and temporal memory safety vulnerabilities. We discuss a series of pointer-based checking approaches developed by Nagarakatte *et al*. [16, 16, 15], which represent state-of-the-art in complete memory safety solutions.

Pointer-based checking, also known as capability systems, treat each pointer as a *capability*, carrying with it an associated set of access rights. Metadata representing these capabilities can be stored inline (as multi-value 'fat pointers' [12]); this approach is common in high-level safe languages, such as Java, and is also used in Cyclone [10], a safe dialect of C/C++. This is often acceptable for new programs, but does not address the problem posed by the huge body of unsafe C/C++ code already in existence, most of which cannot be practically ported to a safe language. Memory integrity checking schemes [24] also protect memory from exploits. However, they take care of direct tampering of the memory (such as physical tampering), and thus are a different direction of security primitives.

Other notable memory safety schemes includes dynamic information flow tracking that tracks all suspicious data in program control flow [25, 20], and context based schemes [2, 24], which associate an identifier with each program data and control entity, and check those identifiers on read and write accesses. However, information flow tracking schemes suffer from high false positive rates occurring due to propagation of normal I/O data into program control flow. On the other hand, context based schemes suffer from high memory requirements and false positives due to pointer interdependencies [9]. Bound checking schemes, such as SoftBound, remove false positives and mitigate memory requirements by creating a capability system based on a disjoint metadata space. Such metadata tracks pointer capabilities in a distinct region of memory

specifically reserved for that purpose [16].

2.1 SoftBound+CETS

SoftBound [16] and Compiler Enforced Temporal Safety (CETS) [17], are complementary software-based capability schemes implemented via compiler-level instrumentation, by adding custom passes to the LLVM compiler infrastructure [13]. In this paper, we refer to SoftBound as state-of-the-art software capability system, with it constituting SoftBound+CETS. SoftBound is based on a previous hardware-based scheme, HardBound [5], and provides complete spatial safety by maintaining a base address and bound corresponding to each pointer. Base and bound values are associated with pointers upon creation, and are propagated to any derived pointers. Each time a pointer is dereferenced, a check is performed to ensure that the effective address is within the allocated region; if it is not, a spatial safety violation is detected and the program is terminated. This approach allows arbitrary pointer arithmetic without compromising safety guarantees. SoftBound can additionally detect sub-object overflows by narrowing pointer bounds when creating pointers to structure members. Type conversion through casts and unions can be performed safely, since the use of a disjoint memory space prevents corruption of metadata. Array and pointer bound metadata is stored and operated upon in a shadow data structure. This is a large data structure that resides in memory, and is convoluted with the normal data of the applications being used. Updates to this structure are quite expensive, requiring multiple loads, stores, and compute operations. SoftBound's performance overhead ranges from negligible to 600%, for various sequential programs [16].

2.2 Memory Safety for Multicores

Previous works have not characterized SoftBound in the context of multicore systems, where several variables, such as synchronization primitives, and cache coherence and consistency, are unique to multicores and present both opportunities and challenges. Metadata security checks within a critical section of a parallel program can significantly degrade performance. Also, metadata accesses over time add to coherence traffic as caches are shared between threads, which further adds overheads. Therefore, multithreading on capability systems present both opportunities and challenges that need to be characterized. Due to SoftBound being a state-of-the-art program control flow protection open-source framework, we use it to analyze and characterize multithreaded applications executing on multicore processors.

3. METHODS

3.1 Multicore Simulation Setup

We evaluate our study using the Graphite shared memory multicore simulator [14]. We consider a 256 core NoC-based multicore organization with a 2-level private L1, shared L2 cache hierarchy per core. The upper limit of 256 for the core count is chosen as it shows all of the effects of scalability. The architectural parameters used for evaluation are shown in Table 1. We use the Graphite simulator because many-core chips with hundreds of cores do not exist yet, whereas Graphite can simulate up to a thousand cores. Graphite relaxes cycle accuracy and uses multi-threading for increased performance. We use both in-order and out-of-order core configurations for analysis. Out-of-order cores can potentially hide latency associated with SoftBound by speculating instructions and memory accesses. When a core misses the L1 cache, an out-of-order core continue to work speculatively, and thus performs computations, such as bound checks in advance, mitigating the performance overheads.

Table 1: Graphite multicore simulator architecture parameters for evaluation.

Architectural Parameter	Value
Number of Cores	256 @ 1 GHz
In-Order Core Setup	
Compute Pipeline per Core	Single-Issue Core
Out-of-Order Core Setup	
Compute Pipeline per Core	Single-Issue Core Out-of-Order Memory
Reorder Buffer Size	168
Load Queue Size	64
Store Queue Size	48
Memory Subsystem	
L1-I Cache per core	32 KB, 4-way Assoc., 1 cycle
L1-D Cache per core	32 KB, 4-way Assoc., 1 cycle
L2 Cache per core	256 KB, 8-way Assoc., 8 cycle Inclusive, NUCA
Cache Line Size	64 bytes
Directory Protocol	Invalidation-based MESI ACKWise$_4$ [11] limited directory
Num. of Memory Controllers	8
DRAM Bandwidth	5 GBps per Controller
DRAM Latency	100 ns
Electrical 2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Contention Model	Only link contention (Infinite input buffers)
Flit Width	64 bits

Table 2: Benchmarks used for Evaluation.

Benchmark	Parellelization	Inputs
Dijkstra [8]	Inner Loop	16384 Nodes
BFS	Graph Division	1048576 Nodes
Prefix Scan	Reduction	16777216 Elements
Matrix Multiply	Tiling	1024x1024 Square
PARSEC [3]	Diverse Parallel	As in specification
SPLASH-2 [29]	Diverse Parallel	As in specification

3.2 Multicore Real Machine Setup

We also evaluate out results on a real machine setup to validate SoftBound slowdowns associated with simulator results. We use an 8 core, 8 thread Intel machine, where each core is an out-of-order processor. Out-of-order cores have better prospects of hiding latencies associated with code dependencies as discussed in the previous section. In the case of SoftBound, latency hiding can be one of the factors in reducing slowdowns, which is observed in subsequent sections.

3.3 Benchmarks

As shown in Table 2, we use several benchmarks to characterize multithreaded bounds checking. The PARSEC [3] and SPLASH-2 [29] workloads are chosen due to their diverse set of programs that can be used to characterize thoroughly. We also use Dijkstra from MiBench [8], and parallel versions of Breadth First Search

(BFS), Prefix Scan, and Matrix Multiply. These workloads are chosen due to their wide usage in practice.

Dijkstra is a memory bound graph workload, it is parallelized via the inner loop, and a dense graph of 16384 nodes and 8192 edges is used. It computes shortest paths from a source node to a destination node in a given input graph. Synchronization is done twice every iteration, which contributes to the communication overhead.

BFS searches an input graph for a particular node. We set this as the final node to cater for worst case searches. Parallel BFS is highly scalable, and allocates a part of the graph to each thread to work on, and a graph of 1048576 nodes, and 16 edges is used.

Prefix Scan computes the sum of the current element and all of its preceding elements. This is parallelized by giving chunks of the input element array to threads. For this workload we use 16777216 elements, and parallelization is done using a tree based reduction strategy.

Parallel Matrix Multiply is computed using tiling on a 1024×1024 sized matrices with floating point entries. The PARSEC and SPLASH-2 workloads did not run to completion with SoftBound, and therefore we only characterize the remaining benchmarks as explained in the subsequent sections. We do however identify the sources of memory safety violations and report them in Table 3.

3.4 Benchmark Characterization

For each simulation run, we measure the *Completion Time*, i.e., the time in *parallel* region of the benchmark, and speedups or slowdowns over sequential versions. This includes the Compute latency including L1 hit latency, the data access latency, and the synchronization latency. For data access latency, we measure latency for the L1 to L2 cache, coherence related latencies, as well as the L2 cache to off-chip latency. Off-chip memory is modeled to incurs a $100ns$ latency as well as the contention delays in the on-chip memory controllers. We run each benchmark compiled with and without SoftBound on the Graphite simulator, as well as the real machine setup.

3.5 Memory Safety Protection

SoftBound is used to protect our benchmarks. We use the latest SoftBound+CETS [17] version that provides complete spatial and temporal safety for C/C++ programs. SoftBound uses the LLVM [13] Clang/Clang++ compiler [27], and is added as a flag in the compilation process. All programs are compiled with `-O2` and `-O3` optimization flags. For the Graphite simulator, programs are compiled with SoftBound, and linked with Graphite's libraries to create executables. These executables are then run on the simulator to analyze results. Some benchmarks use `posix_memalign()` functions to allocate memory, however, SoftBound reports memory safety violations on these function calls. Therefore, these function calls are replaced by `malloc()` functions.

4. CHARACTERIZATION

4.1 PARSEC and SPLASH-2 Benchmarks

We compiled PARSEC [3] and SPLASH-2 [29] benchmarks using the SoftBound Clang compiler. Unfortunately, even though all benchmarks compiled without any issue, none of the benchmarks ran to completion. SoftBound flags the memory safety violation error in the program disassembly, which we use to identify where these vulnerabilities may occur. The main reasons for these violations are out-of-bound access errors, and NULL pointer dereferences. Out-of-bound access violations occur when a pointer accesses an array element that resides outside the array buffer, while NULL pointer dereferences are dangling pointer dereferences. Memory leaks in linked libraries and legacy code stem mostly from dangling dereferences and out-of-bound access violations. Thread-to-thread access violations occur when a thread accesses an element that was allocated for another thread. Table 3 shows an overview of issues for each of these benchmarks. For the PARSEC workloads, most violations occur due to out-of-bound accesses and dangling dereferences, while the SPLASH-2 workloads have a combination of problems. We do not provide specific areas in the programs where the potential security vulnerabilities exist because there might be multiple areas within these programs where these issues may occur.

Table 3: Security issues in PARSEC and SPLASH-2 benchmarks.

PARSEC Benchmarks	Issue
blackscholes	out-of-bound memory access
bodytrack	out-of-bound memory access
canneal	out-of-bound memory access
dedup	NULL pointer dereference
facesim	out-of-bound memory access
ferret	NULL pointer dereference
fluidanimate	out-of-bound memory access
freqmine	out-of-bound memory access
streamcluster	out-of-bound memory access
swaptions	out-of-bound memory access
vips	NULL pointer dereference
x264	NULL pointer dereference
SPLASH-2 Benchmarks	**Issue**
barnes	out-of-bound memory access
cholesky	thread-to-thread memory violations
fft	possible issue in a linked library
fmm	NULL pointer dereference
lu	thread-to-thread memory violations
ocean	NULL pointer dereference
radiosity	NULL pointer dereference
radix	out-of-bound memory access
raytrace	out-of-bound memory access
volrend	thread-to-thread memory violations
water	NULL pointer dereference

4.2 Benchmark Characterization

In this section we discuss detailed performance characterization of the Dijkstra, Prefix Scan, BFS, and Matrix Multiply benchmarks. All evaluations assume a simulated multicore with in-order cores, unless stated otherwise.

4.2.1 Dijkstra Benchmark

Dijkstra, being a memory bound graph workload, constitutes overheads mostly stemming from off-chip accesses, and on-chip data access and communication latencies. The SoftBound version, as shown by the 'S' identifier beside the respective thread count in Fig. 3, adds compute, data access and synchronization latencies. The SoftBound version has a much higher off-chip access rate because the shadow data structure for bounds metadata resides in the off-chip memory, and must be fetched each time a bound check is performed. The bound check also increases the Compute portion because comparisons are performed for every load and store data access.

As more threads are spawned and executed in parallel, the Com-

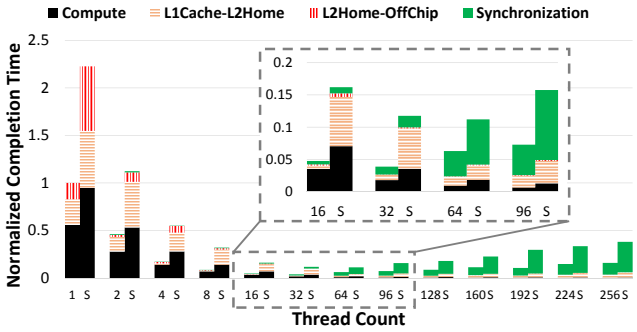

Figure 3: Dijkstra completion time breakdown from Graphite multicore simulator with and without SoftBound. The SoftBound breakdown is shown by 'S' beside the respective thread count.

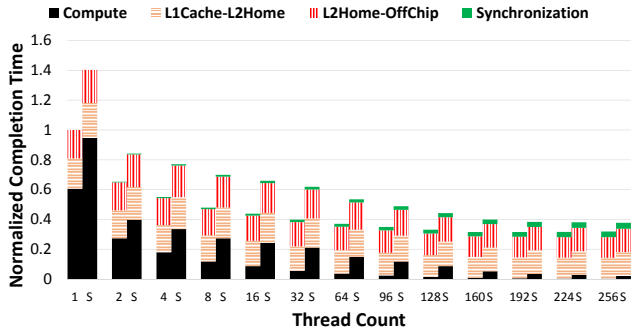

Figure 4: Prefix Scan completion time breakdown from Graphite multicore simulator with and without SoftBound. The SoftBound breakdown is shown by 'S' beside the respective thread count.

pute and data access latency components decrease in both versions due to increase in concurrency. However, the cache communication latencies decrease relatively slowly. Dijkstra scales to ∼32 threads without SoftBound, and 64 threads for the SoftBound version, after which synchronization becomes a bottleneck, since two barriers are applied in each inner loop iteration. This synchronization overhead becomes worse with SoftBound because program threads now wait for longer times at the barriers inside the critical section, waiting for the executing threads to bound check their arrays. Increasing concurrency does help in the case of SoftBound, as completion times for the SoftBound and normal versions match after increasing thread counts. However, there is still no thread count at which the completion times for both versions become equal. Improved security metadata structures can help in this case, as they can potentially have the metadata reside beside the application data. Such optimizations can potentially reduce the data access latency inside critical section, and therefore, also reduce synchronization overheads.

4.2.2 Prefix Scan Benchmark

The parallel Prefix Scan benchmark is chosen due to its lower locality for program data structures, and lesser synchronization effects. As shown in Fig. 4, it constitutes a significant compute and cache communication portion, and a large off-chip portion as well. The off-chip portion remains high due to the metadata shadow space, and decreases slowly with the increase in concurrency, but

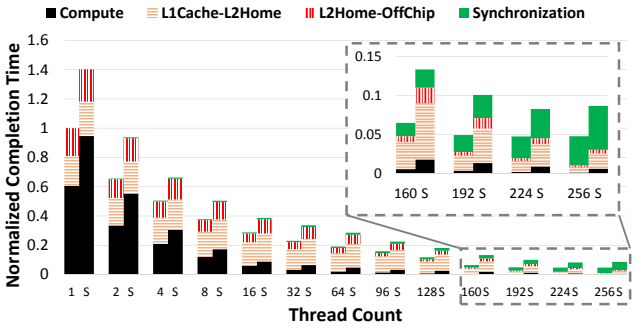

Figure 5: BFS completion time breakdown from Graphite multicore simulator with and without SoftBound. The SoftBound breakdown is shown by 'S' beside the respective thread count.

remains significant due to lower locality of data relative to other evaluated workloads. Large amounts of shared data results in significant L1Cache-L2Home latency, as loads and stores to the array containing the elements is shared amongst all threads. Compute portions scale, while the compute portion of the SoftBound version scales even further with the increase in concurrency. This is because the comparisons required for the bound check get distributed among threads, and have less dependencies. This allows the completion times of the SoftBound version to slowly match those of the version without SoftBound at large thread counts. Synchronization components also slowly rise at large thread counts due to coherency traffic.

Again, increasing the thread count helps reach the same speedup with SoftBound as with the normal version at a lower thread count. Also, at higher thread count, the difference between the normal and SoftBound versions decreases, giving rise to the argument that the overheads of bound checking schemes can be overcome using concurrency. This argument, however, is workload dependent, as we discuss in the subsequent sections.

4.2.3 BFS Benchmark

Another scalable graph workload, BFS, is chosen due to its high locality, and significant compute and communication portions. Both the normal and SoftBound versions scale in Fig. 5, with both compute and off-chip portions being distributed amongst threads. L1Cache-L2Home latency remains significant due to sharing of data structures that determine whether a node has been searched or not. Concurrency helps both normal and SoftBound versions, however, the SoftBound version still lingers behind in terms of speedup. This happens due to fine grain locks in the workload. The SoftBound version's threads have to wait longer to get a lock serviced, and threads keep the lock longer due to metadata bound checks.

4.2.4 Matrix Multiply Benchmark

Matrix Multiply is used pervasively in data analytics, and is therefore chosen as a representative workload. It is both a compute and data bound workload due to floating-point operations on large data inputs. As seen in Fig. 6, both the normal and SoftBound versions scale with the increase in thread count. The compute, off-chip, and cache communication components all reduce with increase in concurrency. However, as the normal version also scales, the SoftBound version lingers behind in terms of speedup. This shows that even scalable workloads suffer from inefficiencies with SoftBound, as observed in Fig. 6. We see that the cache communication portion remains significant even at higher thread counts.

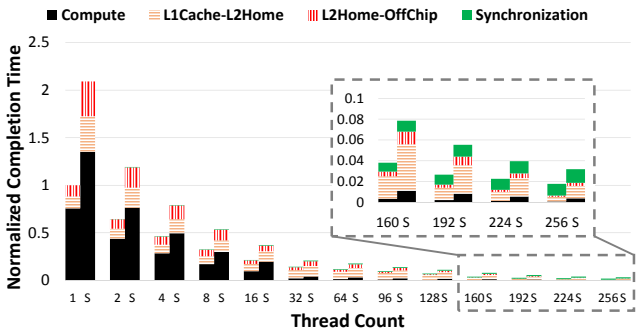

Figure 6: Matrix Multiply completion time breakdown from Graphite multicore simulator with and without SoftBound. The SoftBound breakdown is shown by 'S' beside the respective thread count.

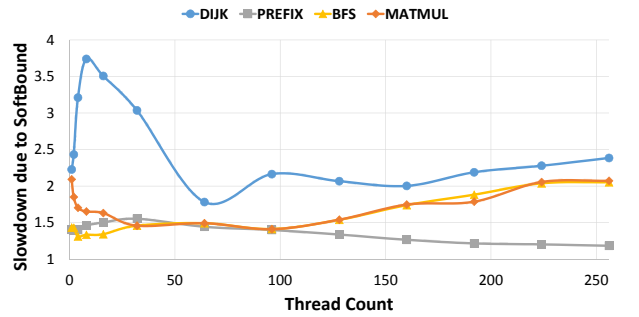

Figure 7: Slowdowns due to SoftBound executing on the Graphite multicore simulator using in-order cores.

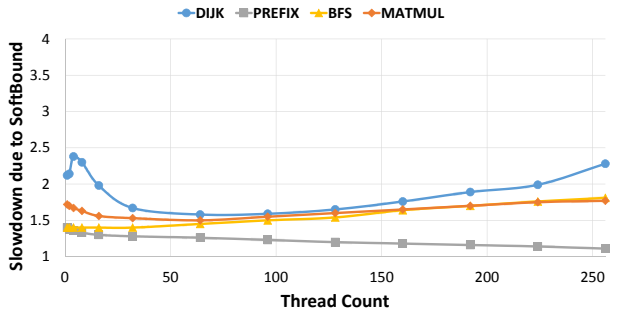

Figure 8: Slowdowns due to SoftBound executing on the Graphite multicore simulator using out-of-order-order cores.

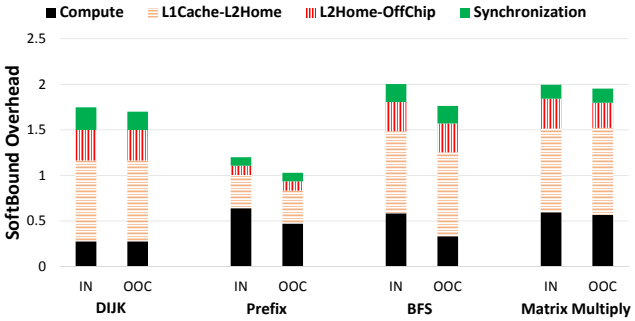

Figure 9: Completion times relative to without SoftBound version. Thread counts with highest speedups are used for comparison.

Data moving around in the cache has to wait until it's bound is checked, which also results in higher synchronization latencies. Low latency and/or high bandwidth off-chip accesses can help with this issue in all the above mentioned benchmarks, and we show one example in the subsequent sections.

4.3 Benchmark Characterization Summary

As evident from Section 4.2, all benchmarks characterized follow different trends. SoftBound versions of Dijkstra and Matrix Multiply initially improve and then worsen with increasing concurrency. The reason mainly being bound checks that increase compute, data access and synchronization overheads. Prefix Scan and BFS improve for SoftBound implementation with increasing thread count. This is because the normal version without SoftBound has low locality and high communication overheads, even at high thread counts. This also provides an insight on how application scalability affects speedups with SoftBound. Fig. 7 summarizes the slowdowns with SoftBound relative to the version without SoftBound. With the exception of Prefix Scan, the slowdown of all benchmarks get worse with increasing concurrency.

4.4 In-Order versus Out-of-Order Cores

As multicores grow larger and denser, more complex and aggressive processing cores can be integrated on chip. Theoretically out-of-order cores can potentially hide the latency associated with SoftBound, as compute and data access operations can be performed in parallel. Fig. 8 shows the results from the Graphite simulator configured to execute out-of-order cores from Table 1. The general trend in slowdowns remain the same. However, the parallel slowdown compared to the in-order core configuration improves for all four benchmarks.

For architectural evaluation, Fig. 9 shows the result for in-order core in comparison to the out-of-order core based multicore. All SoftBound values are normalized to their respective baselines (without SoftBound), and values are taken for the best thread counts (thread counts that gave the highest speedup). Compute times include L1 cache access times as well as core compute times. Out-of-order core results reduce the Compute times for all benchmarks, as speculative accesses can be performed to access security metadata in advance of its execution. Memory bound benchmarks, such as *DIJK* and *Matrix Multiply*, do not show much improvement since most of the overheads are associated with the memory system, whose latencies could not be effectively hidden by the out-of-order cores. Fine grain data dependencies in these benchmarks result in pipeline rollbacks due to miss speculations, and incur their own overheads. Data as well as synchronization latencies are not as much effected as the bound metadata still causes overheads within the memory system. In summary, out-of-order cores do show some improvements in completion time when using SoftBound. However, the memory subsystem still incurs the highest overheads with SoftBound.

4.5 Real Machine Results

We also executed the benchmarks on a real machine setup, and noted the slowdowns due to SoftBound. Real machine setup is run to characterize results for out-of-order cores, smaller cache sizes, and the operating system (OS) overheads. However, the slowdowns still do not show any improvement with the real machine setup, as shown in Fig. 10. The reason for this issue could mainly be the

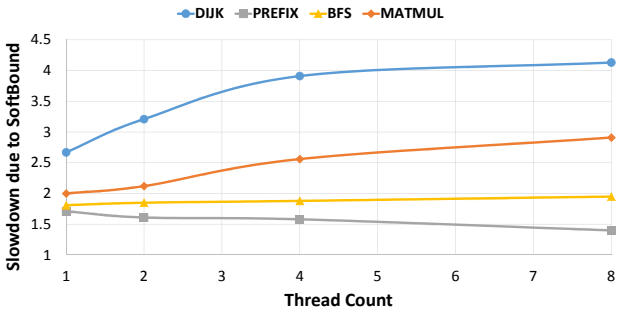

Figure 10: Slowdowns due to SoftBound executing on a real multicore machine setup.

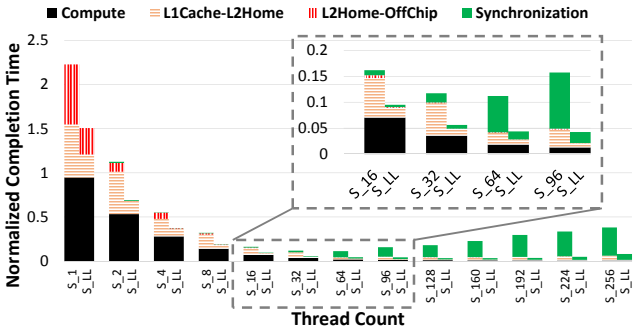

Figure 11: Dijkstra results for a multicore chip configuration with high off-chip bandwidth and low off-chip latency. The SoftBound breakdown is shown by 'S' beside the respective thread count.

memory wall, where SoftBound compiled benchmarks suffer from additional data access and synchronization overheads required for operating on security metadata. The trends remain the same as results gathered from the Graphite simulator in Figs. 7 and 8. Prior bound checking works do not show the multicore characterization of multithreaded benchmarks, and instead focus on sequential benchmarks. As shown in Fig. 7, 8 and 10, performance overheads with SoftBound worsen for most benchmarks, a problem that secure processor researchers must address moving forward.

5. FUTURE WORK

5.1 Data Prefetcher for Security Metadata

The characterization in the previous section has identified that main performance challenge with SoftBound compiled parallel benchmarks is the high data access and communication latency components due to the additional security metadata. Increasing concurrency reduces compute times, however, cache communication and off-chip data accesses still degrade performance significantly. Therefore, to get a better understanding of this problem, we increase each memory controller bandwidth to 10 Gb/second from 5 Gb/second, and reduce the off-chip access latency to $50ns$ from $100ns$ per access. In-order cores are used for this experiment.

We run the Dijkstra benchmark with this setting, for which the results are shown in Fig 11. The SoftBound version performs better than the normal version after a thread count of 64. The reason for this speedup is primarily that the compute, off-chip rates, and the cache communication latencies, are now fully distributed amongst the threads. Metadata from the shadow space can now be quickly retrieved from off-chip into the on-chip cache hierarchy and the requesting cores, allowing for speedy metadata checks. These improvements can be materialized using a novel *data prefetcher* for security metadata. It can prefetch metadata from the off-chip shadow address space in advance of the metadata checks in the processor. Furthermore, latency can be hidden using out-of-order cores. We plan to explore such mechanism as a followup to this paper.

5.2 Smart Security Metadata Structures

One may also improve the memory wall problem by creating an intelligent metadata shadow space. Tree based structures and other compressed versions can be utilized in this context [15]. Such data structures can be intertwined together with the application data, which a novel prefetcher can bring in with the application data into the on-chip cache hierarchy. We plan to also explore smart data structures for security metadata as a followup to this paper.

6. CONCLUSION

We have proposed a solution to hide performance degradation caused by memory safety primitives executing on multicore processors. The solution centers around the fact that hardware concurrency can be exploited in parallel applications to hide performance overheads. Our analysis is presented for several commonly used benchmarks, and we also used both simulator and real machine setups for our evaluation. We argue that given the inevitability of large scale multicores in computing infrastructure, security is a grave need today. Reducing performance overheads can allow the use of program security tools, such as SoftBound, in parallel applications and systems.

7. REFERENCES

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[2] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 165–178, New York, NY, USA, 2014. ACM.

[3] C. Bienia and K. Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[4] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[5] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[6] C. Fletchery, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 213–224, Feb 2014.

[7] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Dec 2001.

[9] B. Hackett and A. Aiken. How is aliasing used in systems software? In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 69–80, New York, NY, USA, 2006. ACM.

[10] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[11] G. Kurian, C. Sun, C.-H. O. Chen, J. E. Miller, J. Michel, L. Wei, D. A. Antoniadis, L.-S. Peh, L. Kimerling, V. Stojanovic, and A. Agarwal. Cross-layer energy and performance evaluation of a nanophotonic manycore processor system using real application workloads. *Parallel and Distributed Processing Symposium*, 0:1117–1130, 2012.

[12] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 721–732, New York, NY, USA, 2013. ACM.

[13] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[14] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010.

[15] S. Nagarakatte, M. M. Martin, and S. Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 175. ACM, 2014.

[16] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[17] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[18] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[19] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[20] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[21] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 17(2):233–263, Mar. 1995.

[22] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *NDSS*. The Internet Society, 2004.

[23] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[24] M. S. Simpson and R. K. Barua. Memsafe: Ensuring the spatial and temporal memory safety of c at runtime. *Softw. Pract. Exper.*, 43(1):93–128, Jan. 2013.

[25] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 85–96, New York, NY, USA, 2004. ACM.

[26] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[27] H. van der Spek, C. Holm, and H. Wijshoff. Automatic restructuring of linked data structures. In G. Gao, L. Pollock, J. Cavazos, and X. Li, editors, *Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, pages 263–277. Springer Berlin Heidelberg, 2010.

[28] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 273–284, Feb 2007.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[30] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting risc in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[31] Y. Younan, W. Joosen, and F. Piessens. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Comput. Surv.*, 44(3):17:1–17:28, June 2012.