Scalable and Efficient Bounds Checking for Large-Scale CMP Environments

Baik Song An, Ki Hwan Yum and Eun Jung Kim
Department of Computer Science and Engineering
Texas A&M University
College Station, Texas 77843-3112
Email: {baiksong,yum,ejkim}@cse.tamu.edu

***Abstract*—Providing spatial safety of memory accesses has been one of the main concerns for the security of programs written in C/C++. Most of the existing approaches to handling spatial memory errors either fail to solve the runtime overhead issues or cannot provide the complete solution for memory protection. Moreover, none of the previous work considers multithread workloads running in multiprocessor systems. In this paper, we attempt to provide an architectural support for fast and efficient bounds checking for multithread workloads in chip-multiprocessor (CMP) environments. Bounds information sharing and smart tagging help to perform bounds checking more effectively utilizing the characteristics of a pointer. Also, the BCache architecture allows fast access to the bounds information. Simulation results show that the proposed scheme reduces the average miss latency of bounds information by 49% as well as reducing μPC of memory operations by 29% on average compared to the previous hardware scheme.**

***Keywords*-architecture; chip-multiprocessor; security; memory attacks; bounds checking;**

I. Introduction

The C/C++ programming languages have been widely used in various programming environments since they were introduced. However, the lack of support for spatial safety of memory addresses in C/C++ has been constantly addressed as one of the major drawbacks. Pointers and array indices must be properly managed to access the memory within their bounds originally assigned to them. If they are allowed to go out of bounds, they might be used as a means of software attacks by accessing unpermitted memory areas. This unsafety can worsen the vulnerability of a system in terms of memory accesses with unchecked pointers or arrays. The problem might become worse when we consider that C/C++ are quite dominant in the system programming area, especially for operating systems and middlewares performing mission-critical tasks.

Moreover, it is obvious that system attacks targeting unsafe memory accesses can be far more complicated and dangerous as multi-core/multi-threaded programming environments become widely adopted in various application domains. In a program running multiple threads, for instance, the value of a pointer locally declared in a thread function might be different since each thread has its own stack. Or, a number of thread-specific pointers might point to one globally accessible memory object by copying a global pointer to multiple thread-specific pointers. It is very common in multi-threaded programs to make global objects shared by threads and synchronize the accesses through mutexes or semaphores. Therefore, it is crucial for chip-multiprocessor (CMP) systems running multiple threads to be equipped with the protection mechanisms that can effectively prevent spatial memory attacks occurring in multi-threaded workloads.

A number of schemes have been proposed so far in order to handle unsafe memory accesses vulnerable to attacks. The most fundamental and simple scheme would be bounds checking that detects whether a pointer value lies within its boundaries when it is dereferenced. The problem that most of the schemes cannot avoid is that they are designed and implemented in the form of software only, which inevitably incurs significant runtime overheads. Some hardware-based schemes that provide architectural supports have been suggested but many of them are specialized for some specific types of memory attacks, not covering the whole ones [1], [2]. HardBound [3] is the first work to provide architectural supports for efficient bounds checking operations and gives a more general solution to protect the system from a wide range of memory attacks. However, it is designed for uniprocessor systems with single-threaded programs, which does not cover the issues of multi-threaded workloads running on CMP systems.

In this paper, we propose an efficient bounds checking mechanism that provides architectural support with marginal performance overheads for multi-threaded workloads running in CMP environments. First we explain how the space overheads can be reduced by implementing bounds information sharing by multiple pointers. Next, smart tagging helps to avoid unnecessary bounds checking that can increase extra overheads. It can be easily observed that a pointer is kept safe most of the time, meaning that its bounds do not need to be checked. We make the best use of these characteristics of a pointer to eliminate unnecessary bounds checking via smart tagging.

Also we attempt to make bounds information needed for bounds checking quickly accessible to each node by designing a new cache and interconnect architecture called *BCache*. BCache allows duplications of the same bounds information to reside in multiple node L2 caches, which reduces the access latency. Performance overheads may increase when updating or invalidating bounds information located in multiple places. But we observe that the bounds information does not change frequently even though its associated pointers keep changing, so the overheads can be effectively managed. We also explore the BCache architecture design in large-scale CMP environments such as locating a bounds node in a cluster under various network topologies and scalable BCache design using 3D stacking. Moreover, BCache can be used to store regular read-only data as well as bounds information.

Simulation results show that the proposed scheme reduces the average miss latency of bounds information by 49% on average compared to the normal coherent cache with the same cache size. Also, our scheme helps to improve the overall performance in terms of μPC by 29% on average

when all the memory operations are executed. We observe that 97% of bounds checking can be skipped through smart tagging, and the sharing of bounds information helps to reduce the space overheads by 9% on average.

The remainder of this paper is organized as follows. We discuss related work in Section II. In Sections III and IV, we explain the efficient bounds checking mechanism and the BCache architecture in detail. Section V presents simulation results and analysis, and finally Section VI summarizes our work and conclusions.

II. Related Work

A. Secure Processor Design

Several schemes have been proposed regarding secure processor design to protect systems from various kinds of attacks. A large portion of them deal with architectural support to prevent physical attacks when data is transmitted through vulnerable parts of the system, mostly off-chip memories and interconnects. To protect the confidentiality of data, each message is encrypted whenever it is injected to an untrusted component. Counter-mode encryption is widely adopted to hide the encryption latency from a critical path. Also for the integrity, an authentication tag is generated using the data value and carried with the message. Caching or predicting counters have been explored to pregenerate pads before encrypted data blocks arrive at the processor from the memory [4], [5], [6]. Also Galois/Counter Mode (GCM) was adopted to accomplish authentication with encryption [7].

New security issues, unique to multiprocessor architectures, began to attract academic interests as the multiprocessor systems became dominant. Rogers, et al. found that there is a strong locality of communication such that one processor communicates with a relatively small number of processors at a time [8]. Lee, et al. proposed I^{2}SEMS in which a global counter controller helps processors predict next counters more accurately [9]. Rogers, et al. proposed a single-level memory protection scheme to reduce the additional security translation overhead incurred by memory controllers [10].

B. Protecting Unsafe Memory Accesses

The protection of spatial memory errors has been explored for a long time in order to prevent memory access violations for programs written in programming languages that do not support boundary checking. A number of schemes based on software approaches were introduced to enforce spatial memory safety. [11], [12], [13] used fat pointer approaches to associating each pointer with its bounds information needed for bounds checking. [14] attempted to make fat pointers using dynamic binary instrumentation. [15] proposed Heap Server to protect the heap metadata. [16] supported shadow values to be used for tracking and detecting dangerous memory accesses. [17] carried out probabilistic analysis to guarantee memory safety using approximated infinite heap. In [18], different software-based implementations of bounds checking were analyzed, and some taint-based optimization techniques were introduced to reduce runtime overheads. Although these studies contributed to prevent spatial memory errors, most of them suffered from huge runtime performance overheads since they were software-based solutions.

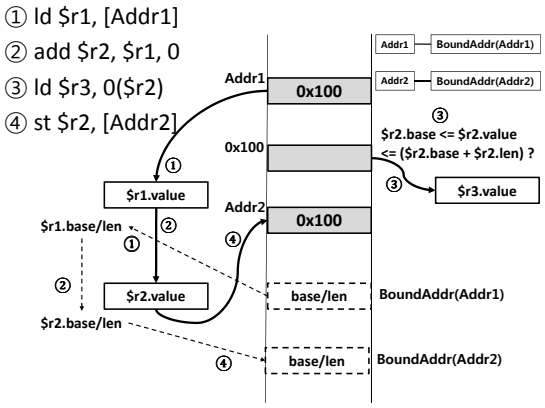


Figure 1: Bounds checking in HardBound

Table I: Expanding a register for bounds checking

Extension	Description
`$r1.value`	Actual value in `$r1`
`$r1.base`	Starting address of a memory object pointed by `$r1`
`$r1.len`	Size of a memory object pointed by `$r1`

To overcome the overhead issues of software-based schemes, making architectural support for memory safety began to be considered. [1] used hardware support to check up the validity of memory block data. [2] provided architectural support of bounds checking for arrays and pointers. However, none of the approaches mentioned above dealt with multiprocessor systems and multi-threaded workloads. HardBound [3] is a hardware-based approach for bounds checking while maintaining the software overheads minimal. It accesses pointers and their associated bounds information with hardware support and propagates them through the system. Also it automatically performs bounds checking before a pointer dereference. But, HardBound always maintains a separate copy of bounds information per each pointer, even when the bounds information can be shared by multiple pointers. Furthermore, it performs bounds checking for all pointer dereferences, no matter whether a pointer value is already proved to be safe or not. The biggest limitation of HardBound in CMP environments is that HardBound does not consider memory hierarchies and interconnects of CMP environments, which may cause significant performance overheads of data accesses for bounds checking.

III. Architectural Support for Efficient Bounds Checking

In this section, we propose two schemes (bounds information sharing and smart tagging) for fast and efficient bounds checking with hardware support. Before going in detail, we explain the basic mechanism of architectural support for bounds checking, which was introduced in HardBound.

All registers that can contain memory address values are expanded to have extra information for bounds checking. Table I describes the extra fields of a register to handle bounds information when a pointer is stored in the register. Figure 1 illustrates a simple example of handling

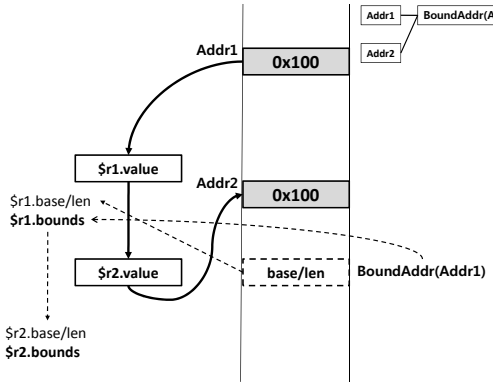


Figure 2: Sharing bounds information

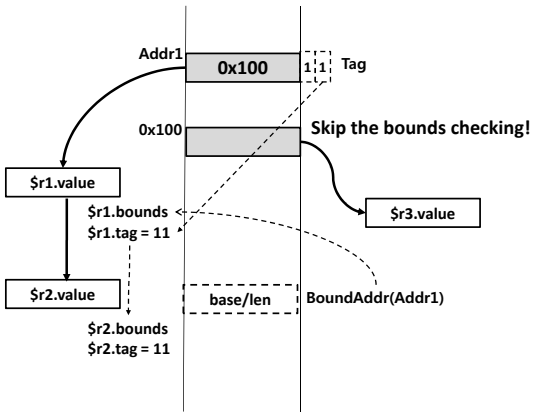


Figure 3: Skipping bounds checking with the smart tagging

bounds information when loading, propagating and storing a pointer. First, a pointer value 0x100 stored in Addr1 is loaded to a register `$r1`. The associated bounds information located in BoundsAddr(Addr1) is loaded to `$r1.base` and `$r1.len`, respectively. Now `$r1.value` has the address value 0x100. When `$r1` is copied to `$r2`, the bounds information stored in `$r1.base/len` is also copied to `$r2.base/len`. When the value in the memory address `$r2.value` (0x100) is loaded to `$r3`, we must make sure that `$r2.value` lies within the bounds, (`$r2.base`) and (`$r2.base + $r2.len`). Finally, the pointer value 0x100 in `$r2.value` is stored to the memory address Addr2. Along with it, `$r2.base/len` are also stored to the address BoundsAddr(Addr2). Note that we omit the bounds checking for Addr1 and Addr2 to simplify the description of the example, even though they must be done actually. The two proposed schemes, bounds information sharing and smart tagging, are explained next.

A. Bounds Information Sharing To Avoid Redundant Storing

Basically, each pointer is associated with its own bounds information in HardBound as shown in Figure 1. The pointers in Addr1 and Addr2 are associated with the bounds information in BoundsAddr(Addr1) and BoundsAddr(Addr2), respectively. However, we can notice that a pointer copied from another pointer has the same bounds information as the original one. This situation occurs quite frequently when a number of pointers in multiple threads point to the same memory object or a pointer value keeps being passed as a parameter in nested function calls. Therefore, memory overheads can be reduced if we allow pointers to share the bounds information.

Figure 2 describes how the bounds information can be shared when a pointer is copied to another. In order to keep track of the location of bounds information when a pointer is propagated through registers, we expand registers to have another extra *bounds* field that contains the address of bounds information associated with a pointer in that register. When a pointer in Addr1 is loaded to `$r1.value`, the address BoundsAddr(Addr1) is loaded to `$r1.bounds` along with `$r1.base/len`. Also when `$r1` is copied to `$r2`, `$r1.bounds` is also copied to `$r2.bounds` together with `$r1.base/len`. When `$r2` is stored to the address Addr2, `$r2.base/len` do not have to be stored since they are the same as the ones in BoundsAddr(Addr1). Instead, we associate Addr2 with the address in `$r2.bounds` that is equal to BoundsAddr(Addr1). Now the pointers in Addr1 and Addr2 share the same bounds information in BoundsAddr(Addr1).

B. Reducing Overheads Using Smart Tagging

Bounds information must be loaded and stored along with its associated pointer. Considering that bounds checking must be performed every time a pointer is dereferenced, we can do the optimization further. First, we do not need to manage bounds information when handling non-pointer values in the system. And more importantly, when a pointer is initialized, it is associated with a memory object for the first time. Then, the pointer is safe because it points to the beginning of that object. Also, a pointer is guaranteed to be safe after passing the bounds checking until it is modified later. In order to perform bounds checking more effectively based on these observations, we use 2-bit tag information per 4-byte memory block in the address space assuming 32-bit ISA. We need a tag for each 4-byte block, since the pointer size is 4 bytes in 32-bit machines. The first bit represents whether the corresponding block has a pointer or a non-pointer; it is set if the value is a pointer and cleared if not. The second bit indicates whether the pointer value stored in the corresponding block is safe or not. It is set if the pointer is safe, when initialized or after passing the bounds checking. It is cleared if the pointer is not guaranteed to be safe, such as after modification. Each register that can contain a pointer is also expanded to have the tag information called `$r1.tag`. The space overhead of the tag information in a 4GB memory address space is 256MB, occupying around 6% of the total space. Note that allocating at least one bit per 4-byte word is mandatory in 32-bit machines for bounds checking with hardware support because it must be figured out whether each word contains a pointer or not to avoid bounds checking for non-pointer values. The smart tagging uses additional one bit per word to

provide a more efficient bounds checking mechanism, which is not a big overhead compared to HardBound that uses up to four bits for each tag.

Figure 3 illustrates how the tag works in the previous example shown in Figure 1. When the value stored in Addr1 is loaded to `$r1.value`, the system first checks up the tag for Addr1. Here both the two bits are set, meaning that the value is a pointer and it is safe. Originally the bounds information should be loaded to `$r1.base/len` if the first bit is set. But it can be deferred until the pointer becomes stale since the second bit is set as well, meaning that the pointer is safe now. The tag is loaded to `$r1.tag` in order to keep track of the safety condition of the pointer. BoundsAddr(Addr1) should be also loaded to `$r1.bounds` because the system must be able to keep track of the location of bounds information when it needs to access the bounds information after modifying the pointer. When `$r1.value` is copied to `$r2.value`, `$r1.tag` and `$r1.bounds` are also propagated along with it. Since the pointer value does not change between the source and the destination registers, `$r2.tag` remains the same as `$r1.tag`. When `$r2.value` becomes different from `$r1.value` later, `$r2.tag` must change to 10 to indicate that the pointer is stale and needs bounds checking before dereference. Also the bounds information must be loaded at this moment. When the value is loaded to `$r3.value` from the address dereferenced by the pointer in `$r2.value`, we can skip the bounds checking as well as loading the bounds information because the pointer is already guaranteed to be safe by looking at the tag in `$r2.tag` that is still 11.

C. Implementation Issues

Caching frequently accessed addresses of bounds information. For faster address translation between pointers and bounds information addresses, each core can be equipped with a bounds TLB that caches frequently accessed mapping information. Using a bounds TLB becomes greatly attractive as workloads show a higher temporal locality of bounds information.

Storing tag information. Unlike the bounds information that can be shared by a number of pointers, the tag information is private to each pointer. Thus, even if multiple pointers share the bounds information, the tags for those pointers cannot be shared. Therefore, tags are to be stored in a normal L1 data cache instead of an L1 bounds cache when accessing the bounds information using BCache architecture that will be explained in more detail in the next section.

Extracting bounds information from a program. In order to get the bounds information at runtime when a pointer is initialized, the program source code can be modified by adding notification every time a memory object is newly assigned to a pointer. It can be done automatically by a source code analyzer or a compiler.

IV. Managing Bounds Information in CMPs

In this section, we explain how to manage bounds information that may be shared by multiple threads in CMP environments. We propose a new cache architecture, which is called Bounds Cache (BCache). BCache allows duplicated copies of bounds information in L2 cache for fast access from threads running on multiple number of cores.

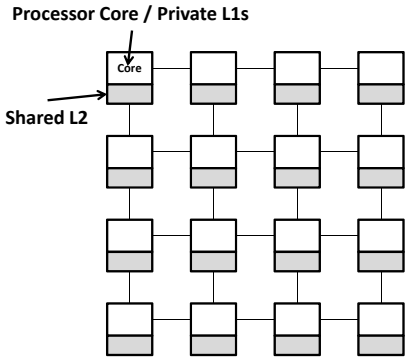
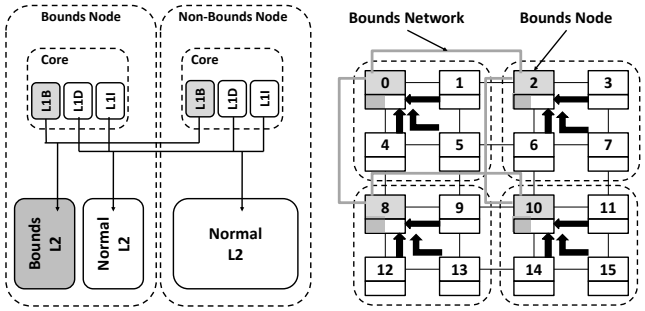


Figure 4: Chip-multiprocessor architecture



(a) Nodes (b) Memory and interconnects

Figure 5: BCache architecture

A. Chip-Multiprocessor Architecture

A CMP integrates more than one processor core onto a single chip. In a CMP, a number of processing elements that can be cores or caches are connected through buses or interconnects. As the system size grows, Network-on-Chip (NoC) is becoming widely adopted as a network to provide the scalability. A router is connected to a processing element or another router through wires transmitting communication data between them. Wormhole switching is commonly adopted in CMP environments to reduce the buffer space needed, and multiple virtual channels are used per each physical channel to avoid Head-of-Line blocking. A network topology determines the way of connecting processing elements and routers, such as a mesh or a fat tree. The CMP architecture assumed in this paper is a tiled CMP in which tiles are connected through a mesh network. Each tile consists of a core with private L1 caches and a part of shared L2 caches including directory information. Figure 4 shows a tiled 16-core CMP architecture.

B. BCache Architecture in CMP

Figure 5 describes the overall architecture of BCache for a 16-core CMP system. As seen in Figure 5(a), additional L1 bounds cache (L1B) is used along with the existing L1 instruction/data caches (L1I/L1D) to manage and access

bounds information fast and efficiently. When bounds information is located in the L2 cache of BCache architecture, only some specific nodes can have the bounds information in their L2 caches, not allowing other nodes to store it, which is different from the normal shared L2 cache that allows bounds information to be placed in any node depending on the address. Four out of sixteen nodes, 0, 2, 8 and 10 in Figure 5(b) are chosen and they are called *bounds nodes*. Bounds nodes have a small part of their L2 caches dedicated to store bounds information and use the other part of L2 for normal data. The other non-bounds nodes have all their L2s only for normal data. Unlike the other L1 instruction/data caches, the L1 bounds cache is write-through, thus newly written bounds data in the L1 bounds cache is immediately written to the L2 cache of a bounds node as well. It enables fast access to new bounds information from other nodes. For fast transmission of bounds information between bounds nodes, a separate interconnection network is adopted only for bounds data, and we call it *bounds network* hereafter. The bounds network connects all four bounds nodes in a mesh style as depicted by gray lines in Figure 5(b) and allows communication between bounds nodes within two hops. The normal mesh network is used for other communication including normal data traffic as well as the communication between bounds and non-bounds nodes.

The most important difference between BCache and normal shared L2 cache architectures is that each bounds node can have its own duplicated copy of the same bounds information in BCache, whereas normal shared L2 cache allows only one copy in the system. To keep track of the duplication information, each cache block in L2 of BCache has extra 4-bit information indicating whether each of four bounds nodes has the block in its L2. All bounds nodes having duplicated copies of a bounds data block have the same duplication information for that block. Whenever the bounds data block is duplicated or evicted in a bounds node, all bounds nodes sharing that data updates the duplication information. In order to make a fast access to the bounds nodes from non-bounds nodes, one bounds node and three of its neighboring non-bounds nodes are grouped together and the group is called a *bounds cluster* as shown in Figure 5(b). When a miss occurs in L1 bounds cache of a node, a request message is first sent to the corresponding bounds node in its bounds cluster. Note that the bounds node is located close to other nodes in the cluster, which decreases the latency of a first miss request message and a data message between the requestor and its bounds node. In the normal shared L2 architecture, the distance between the requestor and the owner that provides the data may be quite far. When the bounds node receives the request message, it immediately replies to the requestor if it has the data. Or it can communicate with other three bounds nodes to get the data from them. All four bounds nodes efficiently share and transfer bounds information among them, which will be explained in more detail in Section IV-C.

C. Handling Bounds Information in BCache

Figure 6 illustrates the examples of how the bounds information can be managed by the BCache architecture in various situations.

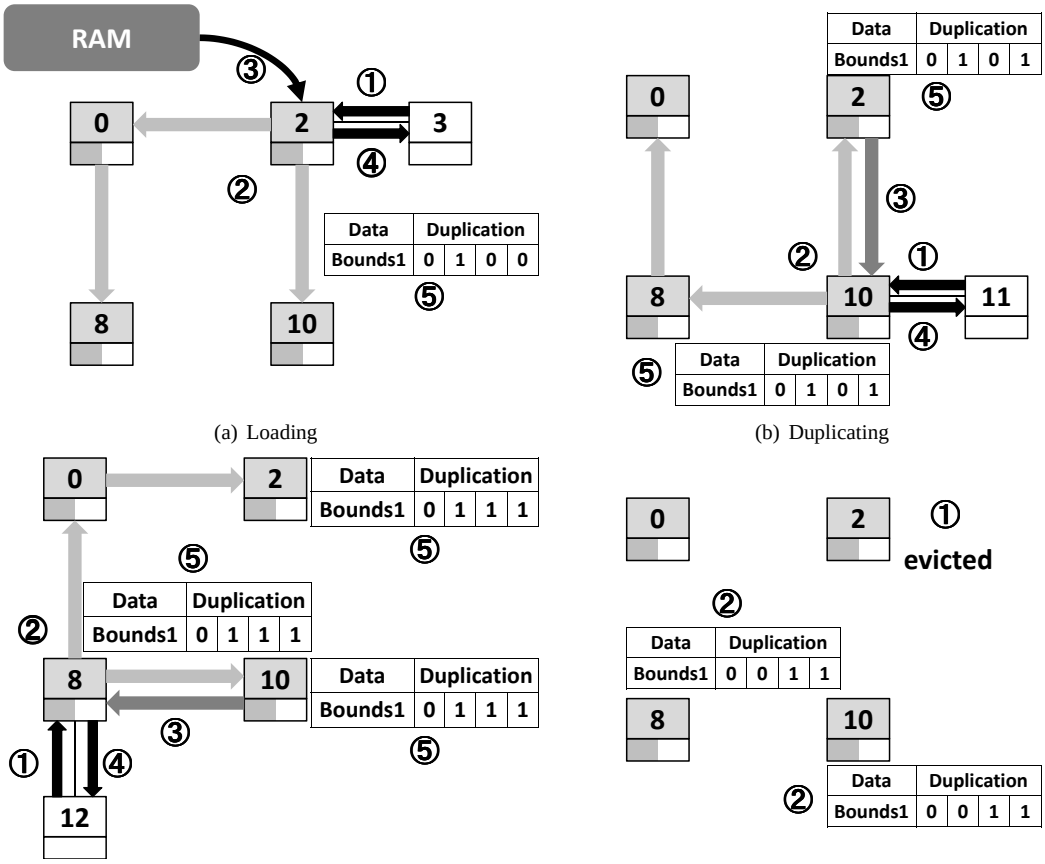
Loading bounds information. Suppose that a cache miss occurs in the L1 bounds cache of node 3 as shown in Figure 6(a). First, node 3 sends a request message to the bounds node in the cluster, node 2 in this example. If node 2 has the requested data in its L2, it provides the data to the requestor. Otherwise, it requests other three bounds nodes (nodes 0, 8 and 10) to figure out whether any of them has the data through the bounds network. [1] If the data is found in one of the other three bounds nodes, it is transferred to node 2. Or it can be fetched from an off-chip memory if none of them has it. Finally, node 2 has the requested data and sends it to the requestor, node 3, using the normal mesh network. As explained in the previous section, each cache block in bounds nodes maintains the 4-bit duplication information to keep track of the duplication status of bounds data. Thus, node 2 updates the duplication information for the current data block. If any other node in the cluster such as node 7 requests the same data later on, the node 2 can immediately provides the data from its own L2.

Loading the same bounds information in a node of a different cluster. Figure 6(b) shows how the bounds information previously requested by node 3 is provided to node 11 in a different cluster. Node 11 first sends a request message to the bounds node in the cluster, node 10, as before. If node 10 does not have the requested data, it requests other three bounds nodes to search for it. Since the bounds node 2 has the data, it sends the data to node 10. Requesting and receiving data between bounds nodes are done through the bounds network as in the previous example. After receiving the data from node 2, node 10 sends the data to the requesting node 11. Now the bounds nodes 2 and 10 have duplicated copies of the same bounds data in their L2. Here both the nodes 2 and 10 have the knowledge of the bounds data block duplicated in those two nodes.

Loading the bounds information after thread migration. Threads running in the system keep migrating to different processor cores based on the OS scheduler's policy. Then the same bounds information may be requested from a new node after the migration finishes. As shown in Figure 6(c), if the thread running on node 3 is migrated to node 12, node 12 can request the bounds information that was originally used by node 3. Getting the bounds information and updating the duplication information can be done as explained above. Note that node 12 can make a fast access to the bounds information with the help of BCache architecture, even though nodes 3 and 12 are located far from each other in terms of hop counts in a normal mesh network.

Evicting the bounds information from a bounds node. If a new bounds data is written to the L2 cache in a bounds node when the cache set is full, one bounds cache block must be

[1] If the L2 of a bounds node does not have the data, the corresponding duplication information is not available in that node as well. So node 2 has no information of which bounds node has the data.



(a) Loading (b) Duplicating

(c) Duplicating after migration (d) Evicting

Figure 6: Managing bounds information in BCache

chosen to be evicted based on the LRU replacement policy. In Figure 6(d), when a bounds cache block is duplicated in the bounds nodes 2, 8 and 10 and the one in node 2 is evicted, node 2 sends the update message to the other two bounds nodes 8 and 10 to update the duplication information. Once nodes 8 and 10 gets the updates message, they delete node 2 from the duplication information. If the evicted block is the only copy and it is not stored in the off-chip memory yet, it is written back to the memory after eviction.

Modifying the existing bounds information. The bounds information can be modified at runtime by `realloc()` and so on. If it is modified, all the previous copies of that bounds information must be invalidated. Suppose that the `realloc()` is called in node 3 to modify the bounds information. Right after a new bounds data is stored in the L1 bounds cache of node 3, it is immediately sent to the bounds node 2 for update since the L1 bounds cache is write-through as explained in Section IV-B. Then the bounds node 2 broadcasts the new bounds data to all other three bounds nodes to invalidate the old data through the bounds network. Once each bounds node receives the invalidation message, it updates the duplication information and broadcasts the message again to its neighboring nodes in its cluster through the normal mesh network for invalidation. After invalidation, the new bounds data is located only in the bounds node 2, and getting the new data from other nodes can be done in the same way as explained above. This broadcast-based invalidation procedure may increase the overheads compared to the normal cache coherence protocol, since an invalidation message must be broadcast to all nodes every time a write occurs. But usually updating bounds information is very rare in a program, so it will not worsen the overall performance much.

D. Scalable Design of BCache Architecture

When a CMP system scales up to have more processor cores in a chip, each bounds node must be properly located in a cluster for optimal results. Thus, for each node, we calculate the average distance to reach all nodes in a cluster. Suppose that an axb bounds cluster is placed on an imaginary X-Y graph. Then each node corresponds to one of the coordinates on the graph such as $(0, 0)$ and $(a-1, b-1)$. Assuming that C is the set of all cluster nodes, average distance from an arbitrary node $p = (x, y)$ to all nodes in a cluster can be obtained as follows.

$$AvgDist_p = \frac{\sum_{(x_1,y_1)\in C}(|x - x_1| + |y - y_1|)}{a \times b}$$

We can generalize this formula to a mixed radix $k_0 \times k_1 \times \cdots \times k_{n-1}$ n-mesh network. Here the distance between $p = (p_0, p_1, \cdots, p_{n-1})$ and $q = (q_0, q_1, \cdots, q_{n-1})$ is as follows.

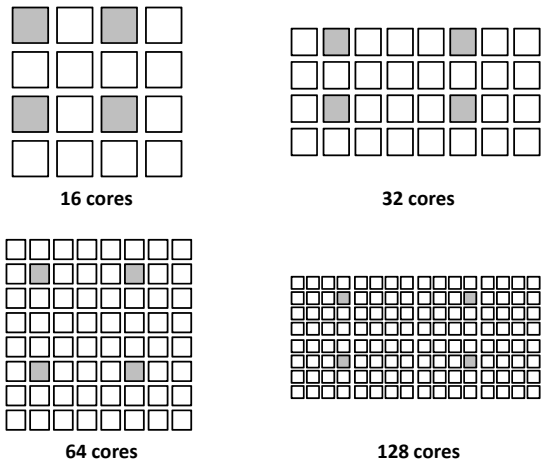


Figure 7: Scalable BCache design in CMP

$$Dist(p,q) = \sum_{i=0}^{n-1} |p_i - q_i|$$

The average distance from an arbitrary node $p = (p_0, p_1, \cdots, p_{n-1})$ to all nodes in the cluster is defined as follows.

$$AvgDist_p = \frac{\sum_{q \in C}(Dist(p,q))}{\prod_{i=0}^{n-1} k_i}$$

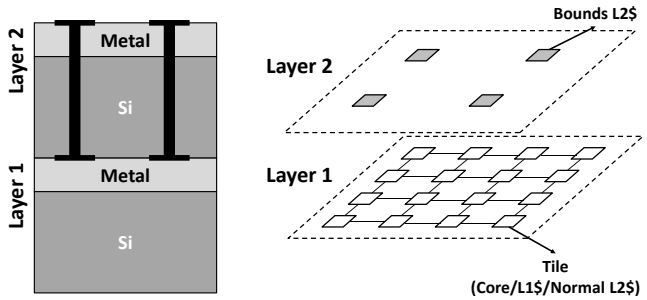
Using this formula, we choose the place of a bounds node that minimizes average distance. Figure 7 illustrates the BCache architectures deployed in 4x4, 8x4, 8x8 and 16x8 2-D mesh CMP systems.

In the case of the concentrated mesh (CMesh) topology [19] in which routers are connected in a mesh style and each router services four nodes, we can treat a number of nodes sharing a router as a group. Then, the topology becomes a normal mesh connecting those groups. Here we can pick up the group with the shortest average distance using the formula shown above. Then we can choose any node in that group as a bounds node since all nodes have the same distance from the router.

E. Design Alternatives for Large-Scale BCache Architecture

When the system size grows in a large scale, placing bounds nodes in traditional mesh-style topologies has a limitation on reducing the latency to reach a bounds node from a normal node. As an alternative, three-dimensional (3D) die-stacked architecture might be adopted to shorten the latency in a more efficient way. In the 3D architecture, multiple silicon dies are stacked together and they communicate each other through vertical interconnects. Figure 8(a) describes a 3D architecture with two layers connected with Face-to-Back (F2B) bonding using Through Silicon Vias (TSVs) .

BCache architecture can be designed using two different silicon layers as shown in Figure 8(b). The first layer contains nodes with cores, L1 caches and normal L2 caches connected with a normal mesh network. Bounds nodes connected with the bounds network are located in the second



(a) Face-to-Back (F2B) bonding

(b) 3D design of BCache architecture

Figure 8: Design alternative using 3D stacking

layer. Since each node is connected to the bounds node via TSV, the access delay from all nodes to the bounds nodes becomes one hop regardless of the system size. Too many vertical interconnects might take up much space and compete with other devices for the area budget. However, it is not that problematic in our scheme because we only put the bounds nodes in the upper layer that do not consume the area much.

Locating bounds nodes in a separate silicon layer allows more flexible design of the bounds network since an area budget becomes bigger. For example, we can make all vertical connection points in the upper layer connected to a high-speed media such as an optical shared bus in the manner of a Hamiltonian path [20]. Using a bus-style shared media might cause scalability problems when highly congested, but accessing bounds data usually occurs quite rarer than normal memory access for instructions and data. The high-speed shared bus also helps to communicate between bounds nodes when transmitting bounds data or updating the sharing information.

F. Using BCache for General Purposes

One may consider BCache is too costly to be used exclusively for bounds checking. Here we show how BCache can be used more generally.

In addition to bounds information, BCache can be used to store regular data accessed by normal load and store instructions. Since BCache is not efficient in handling heavily overwritten data due to the invalidation overheads, read-only data would be a good candidate for BCache. To do that, the source code should be annotated for those data as in bounds information. It can be done in a number of ways. For example, a source code analysis tool such as [21] can be used to detect read-only variables and data objects in the program code. Or the information stored in a binary executable file can be exploited to obtain read-only address spaces.

Also, BCache can be reconfigured to be a part of normal cache hierarchy when the bounds checking is not necessary. Bounds caches in L1 and L2 can work as a part of normal L1 data or L2 caches by reconfiguring the L1 bounds cache from write-through to write-back and assigning a portion of physical memory address space to BCache exclusively.

Table II: System parameters

Simulators	Parameters	Values
Simics/ FeS_2	CPU	16 Intel x86 processors
	Cache block size	64B
	Memory	1GB, 300 cycle access time
	Directory access time	80 cycles
	OS/Kernel	Fedora Core 5 (x86) / Linux 2.6.15
CMP CacheSim	L1 cache	4-way, 1KB, 2 cycles HardBound: Write-back BCache: Write-through
	L2 cache	4-way, 1MB, 4 cycles HardBound: 128K x 16 BCache: 256KB x 4
	Cache block size	64B
	Memory	1GB, 300 cycle access time
	Router	Fixed 5-cycle pipelined router
	Network topology	4x4/8x4/8x8/16x8 mesh

If the memory address belonging to that address range is accessed by a load or store instruction, L1 bounds cache is looked up instead of normal L1 data cache. A cache miss in L1 bounds cache is handled by one of L2 bounds cache banks depending on the address. Note that L2 bounds caches do not allow data duplication in this case, so each cache block can be stored in only one L2 bounds cache bank. To maintain cache coherence, each cache line in L2 bounds cache should have directory information in addition to the existing 4-bit sharing information. As in the bounds checking, packet traverse between bounds nodes can be done via bounds network. Also, the traverse between bounds and non-bounds nodes is done via normal mesh network. The configuration of L1 bounds cache and the memory address range can be supported by system BIOS.

V. Performance Evaluation

A. Simulation Framework

We measure the performance of the proposed schemes using Simics full-system simulator [22]. In order to simulate the bounds checking mechanism explained in the previous sections, we also use a Full-system Execution-driven Simulator for x86 (FeS_2) that models out-of-order x86 processor cores including a decoder used to convert an x86 instruction to RISC-style micro-ops. FeS_2 integrates PTLsim [23] decoder and Ruby memory model in GEMS [24]. To measure the performance of BCache architecture, we capture all memory reference traces including bounds information using Simics and FeS_2. Then those traces are fed to a cycle-accurate CMP cache simulator that models three different cache architectures: HardBound, BCache and BCache with 3D stacking. Note that HardBound is designed for uniprocessor systems, so it does not assume a specific cache coherence protocol. So we assume a general directory-based MSI coherence protocol for HardBound, while BCache uses its own protocol explained in Section IV-C.

Table II shows the system parameters used in the simulation. To make a fair comparison, the total size of bounds L2 cache in BCache is configured the same as that of L2 cache of HardBound. Also, we assume that both HardBound and BCache access bounds information through 1KB L1 bounds cache, even though HardBound actually stores the bounds information in L1 data cache.

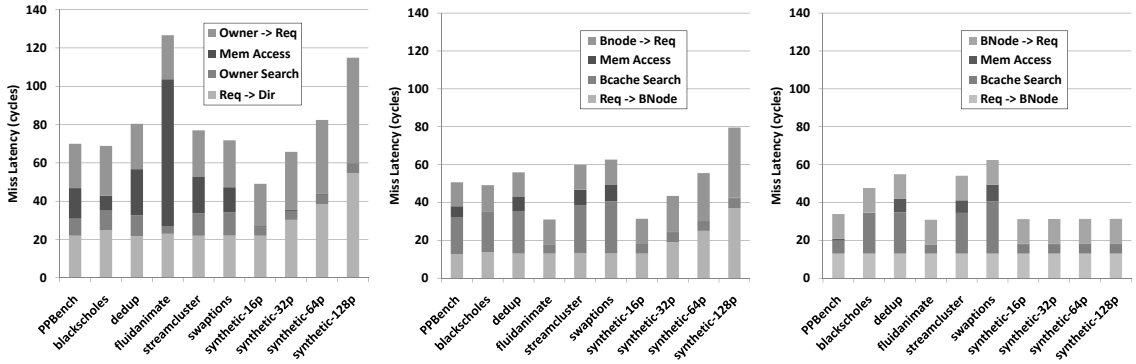
We use five PARSEC [25] (*blackscholes, fluidanimate, dedup, streamcluster, swaptions*) benchmarks with `simsmall` input sets for parallel workloads. Also we make our own parallel benchmark called *ParallelPointerBench* to measure the performance with more pointer-intensive workloads. ParallelPointerBench spawns threads as many as the number of cores and generates total 2,000 global and thread-specific pointers. Each thread accesses pointers 2,000 times at random. All benchmark source code has been modified to extract read-only data and bounds information for pointers by adding a simple notation. Note that we evaluate only the benchmark code itself, not libraries and OS kernel.

In order to measure the scalability of BCache architecture, memory reference traces are needed for systems with more than 16 processor cores. However, configuring and running Simics for large-scale CMP systems requires huge amount of time. So we could not use Simics to generate traces for large-scale systems. Instead, we make a trace generator that is able to create traces for arbitrary number of cores. Using that generator, traces have been made that simulate workloads generating and accessing pointers randomly for 16, 32, 64 and 128-core systems. The ratio of writes to read operations is set to 3% in each trace.

B. Simulation Results

First, we clarify how much the BCache architecture is beneficial in handling bounds information in CMP systems. Figure 9 shows average miss latencies of loading and storing bounds information for HardBound, BCache and BCache-3D designs. In each graph, the first column *PPBench* represents ParallelPointerBench and the next five columns shows PARSEC benchmark results. The last four columns show the results obtained from synthetic traces created by the trace generator we made for 16, 32, 64 and 128 cores. To compare the results in more detail, we break down the latency values into four parts for each cache design. In HardBound shown in Figure 9(a), the miss latency consists of the request latency from a requestor to the corresponding directory node, the lookup latency needed to reach the owner, the off-chip memory access latency in the case of L2 miss, and the latency of data transfer from the owner to the requestor. Similarly, the miss latency of BCache and BCache-3D designs in Figure 9(b) and 9(c) consists of the request latency from a requestor to its bounds node, the lookup latency for fetching data among four bounds nodes and placing it in the requested bounds node, the off-chip memory latency and the data transfer latency from the requested bounds node to the requestor.

BCache improves the miss latency by 36% on average compared to HardBound. It is noteworthy that BCache achieves significant performance enhancement in terms of the first request and the final data transfer latencies since the bounds node providing the data is located close to the requestor in BCache. Using BCache with 3D stacking, the miss latency is reduced much further by 49% in overall. The amount of improvement increases significantly as the system size grows, up to 73% in the case of a 128-core mesh-style



(a) HardBound (b) BCache (c) BCache-3D

Figure 9: Average miss latencies of accessing bounds information

CMP system. 3D stacking contributes to this improvement by allowing all nodes in the lower layer to reach its bounds node in one hop through the vertical interconnect. We fix total number of accesses to the bounds information for all different system sizes, which results in the decrease of an injection rate per node as the system size grows. Therefore, the results in this experiment do not include any delay caused by contention as the network size grows. If the injection rate per node is kept constant as the system scales up, we may observe some congestion delay in communication using bounds network. For this study, we focus on the delay caused by topologies. Another meaningful result is the reduced off-chip memory access latency in BCache. BCache places duplicated copies of the bounds cache block in multiple bounds nodes in a smart way, which reduces the number of L2 misses in bounds nodes. As shown in the results from synthetic traces, BCache architecture is quite scalable showing the better performance even in large-scale systems.

We also measure the average miss latencies for regular read-only data. Here we do not show the results from BCache with 3D stacking since BCache with 3D stacking shows the results quite similar to BCache under a 16-core system in terms of the average miss latency as shown in Figure 9. In Figure 10, BCache outperforms HardBound again by 51% on average. BCache is designed to handle read-only data efficiently as well as bounds information, while HardBound assumes general memory and cache coherence models with no optimization. So HardBound does not show the performance improvement for read-only data.

Next, we investigate how BCache and the smart tagging affect the overall system performance. We use traces of read-only data and bounds information together and apply the smart tagging to skip bounds checking. Figure 11 shows the performance of three different schemes in terms of micro-ops per cycle(μPC); HardBound, BCache and BCache with skipping bounds checking. Here we use μPC instead of IPC as a performance metric since each x86 instruction is decoded to a number of micro-ops as explained before. We do not show the results from large-scale systems as well as BCache with 3D stacking for the same reason in Figure 10. Figure 11 shows μPC of all memory operations executed. Using BCache and the smart tagging improves the

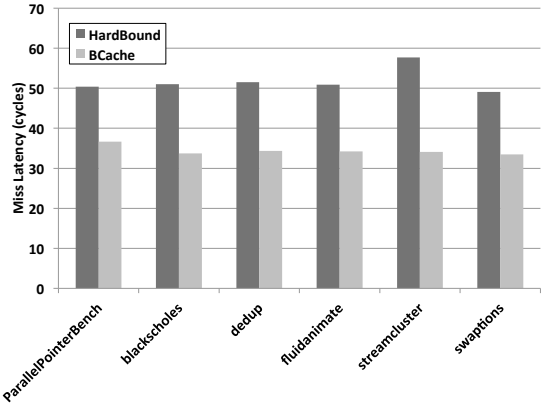


Figure 10: Average miss latencies for read-only data

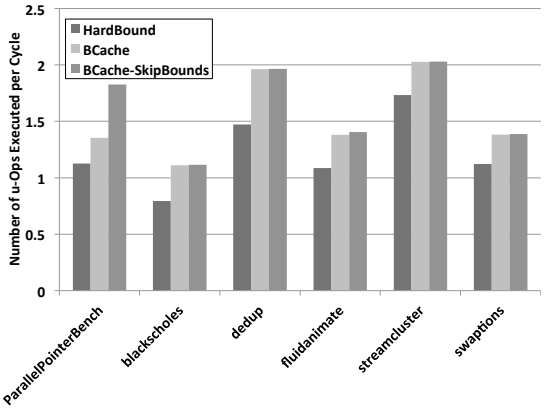


Figure 11: Performance improvement of memory accesses

performance by 29% on average compared to HardBound. Note that we handle only application code, not libraries and OS kernel. Therefore, the amount of improvement will increase significantly if the bounds checking can be done for all those codes. Here the performance improvement of skipping bounds checking looks somewhat marginal. Since PARSEC benchmark suite is not pointer-intensive, the cache traffic of bounds information is quite sparse compared to that of all memory operations. A noticeable performance improvement in the pointer-intensive ParallelPointerBench

Table III: Number of bounds checking

Benchmarks	Total	Skipped	Percentage
PPBench	296907	295954	99.679024%
blackscholes	3247416	3247395	99.996921%
dedup	53458	46598	87.167496%
fluidanimate	812156	809873	99.718896%
streamcluster	522627	519172	99.338917%
swaptions	11840543	11834754	99.951109%

Table IV: Storage overheads

Level	HardBound	BCache
L1	2KB/8KB per node (tag cache)	1KB per node (L1 bounds cache)
L2	use normal L2 cache	256KB x 4, a part of total L2 cache (L2 bounds cache)

supports our analysis. To verify how much our scheme is beneficial in skipping bounds checking in more detail, we show the ratio of skipped bounds checking with the smart tagging out of total bounds checking in Table III. We can see that approximately 97% of bounds checking can be skipped on average.

We clarify how much the space overhead can be reduced through the sharing of bounds information among multiple pointers. Figure 12 shows the amount of address space allocated to the bounds information. The first bar in each column represents the result when bounds information is not shared. The second bar depicts the result of shared bounds information normalized to the first one. We can verify that 9% of space overheads can be reduced on average when pointers pointing to the same object shares the bounds information. Finally, we compare the extra overheads of HardBound and BCache for storing bounds information in Table IV. The overheads of BCache is 25% to 50% of the overheads of HardBound. Therefore, BCache does not increase the overheads much, compared to HardBound.

VI. Conclusions

In this paper, we have proposed an architectural support for fast and efficient bounds checking for multi-threaded workloads in CMP environments. We reduce the space overheads through bounds information sharing as well as adopting smart tagging that enables the skipping of bounds checking for pointers already guaranteed to be safe. The BCache architecture allows fast delivery of bounds information as well as regular read-only data to a requestor node by duplicating the same data block that might be shared by threads in multiple locations. Simulation results show that the proposed scheme reduces the average miss latency of bounds information by 49% as well as reducing μPC of memory operations by 29% on average. The memory space allocated for bounds information reduces by 9% on average compared to HardBound.

Our work can be explored further by investigating the address mapping mechanism of bounds information in more detail. Also we plan to examine the BCache design for other topologies such as fat tree or flattened butterfly.

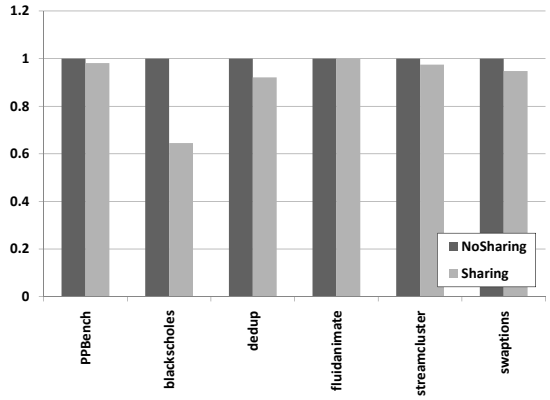


Figure 12: Effect of sharing bounds information

References

[1] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging," in *Proceedings of HPCA*, 2007.

[2] Z. Shao, J. Cao, K. C. C. Chan, C. Xue, and E. H.-M. Sha, "Hardware/Software Optimization for Array & Pointer Boundary Checking Against Buffer Overflow Attacks," *Journal of Parallel and Distributed Computing*, September 2006.

[3] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural Support for Spatial Safety of the C Programming Language," in *Proceedings of ASPLOS*, 2008.

[4] G.E.Suh, D.Clarke, B.Gassend, M. Dijk, and S.Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," in *Proceedings of MICRO*, 2003.

[5] J.Yang, Y.Zhang, and L.Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," in *Proceedings of MICRO*, 2003.

[6] W.Shi, H.S.Lee, M.Ghosh, C.Lu, and A.Boldyreva, "High Efficiency Counter Mode Security Architecture via Prediction and Precompuatation," in *Proceedings of ISCA*, 2005.

[7] C.Yan, B.Rogers, D.Englender, Y.Solihin, and M.Prvulovic, "Improving Cost, Performance, and Security of Memory Encryption and Authentication," in *Proceedings of ISCA*, 2006.

[8] B.Rogers, M.Prvulovic, and Y.Solihin, "Efficient Data Protection for Distributed Shared Memory Multiprocessors," in *Proceedings of PACT*, 2006.

[9] M.Lee, M.Ahn, and E.J.Kim, "I^{2}SEMS: Interconnects-Independent Security Enhanced Shared Memory Multiprocessor Systems," in *Proceedings of PACT*, 2007.

[10] B.Rogers, C.Yan, S.Chhabra, M.Prvulovic, and Y.Solihin, "Single-Level Integrity and Confidentiality Protection for Distributed Shared Memory Multiprocessors," in *Proceedings of HPCA*, 2008.

[11] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Code," in *Proceedings of POPL*, 2002.

[12] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient Detection of All Pointer and Array Access Errors," in *Proceedings of PLDI*, 1994.

[13] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly Compatible and Complete Spatial Memory Safety for C," in *Proceedings of PLDI*, 2009.

[14] N. Nethercote, "Bounds-Checking Entire Programs Without Recompiling," in *Proceedings of SPACE*, 2004.

[15] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and Efficiently Protecting the Heap," in *Proceedings of ASPLOS*, 2006.

[16] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of PLDI*, 2007.

[17] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic Memory Safety for Unsafe Languages," in *Proceedings of PLDI*, 2006.

[18] W. Chuang, S. Narayanasamy, B. Calder, and R. Jhala, "Bounds Checking with Taint-Based Analysis," in *Proceedings of HiPEAC*, 2007.

[19] J. Balfour and W. J. Dally, "Design Tradeoffs for Tiled CMP On-Chip Networks," in *Proceedings of ICS*, 2006.

[20] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. P. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. G. Beausoleil, and J. H. Ahn, "Corona: System Implications of Emerging Nanophotonic Technology," in *Proceedings of ISCA*, 2008.

[21] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Proceedings of the CC*, 2002.

[22] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[23] M. T. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *Proceedings of ISPASS*, 2007.

[24] M.M.Martin, D.J.Sorin, B.M.Beckmann, M.R.Marty, M.Xu, A.R.Alameldeen, K.E.Moore, M.D.Hill, and D.A.Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News(CAN)*, 2005.

[25] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of PACT*, 2008.