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*—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 increases 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. 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. A number of schemes have been proposed so far in order to handle unsafe memory accesses vulnerable to attacks, but none of them was not successful in preventing memory attacks, especially for 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.

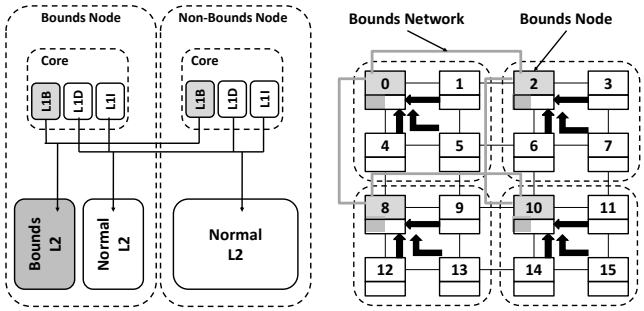
II. Architectural Support for Efficient Bounds Checking

We propose two schemes (bounds information sharing and smart tagging) for fast and efficient bounds checking with hardware support. First, all registers that can contain memory address values are expanded to have associated bounds information for bounds checking. In Table I, *base* and *len* depict the starting address of a memory object and the object size, respectively. All bounds information is automatically handled by hardware whenever a pointer is loaded, propagated and stored.

Bounds information sharing. Note that a pointer copied from another pointer has the same bounds information as the original one, which 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. In this case, memory overheads can be reduced if we allow pointers to share the bounds information. So we keep track of the location of bounds information when a pointer is propagated through registers.

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`
`$r1.bounds`	Address of bounds information of a pointer in `$r1`
`$r1.tag`	Tag information of a pointer in `$r1`



(a) Nodes (b) Memory and interconnects

Figure 1: BCache architecture

Smart tagging. Considering that bounds checking must be performed every time a pointer is dereferenced, we can perform 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 32-bit ISA to represent whether the corresponding block has a pointer and the pointer value is safe.

To implement our schemes, each register is expanded

further to have extra information about the location of bounds information and the tag, called *bounds* and *tag* in Table I. A more detailed explanation can be found in [1].

III. Managing Bounds Information in CMPs

We propose a new cache architecture, which is called Bounds Cache (BCache). BCache allows duplicated copies of bounds information in L2 cache for fast accesses from threads running on multiple number of cores.

Figure 1 describes the overall architecture of BCache for a 16-core CMP system. As seen in Figure 1(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 the BCache architecture, only some specific nodes called *bounds nodes* can have the bounds information in their L2 caches, not allowing other nodes to store it. For fast transmission of bounds information between bounds nodes, a separate interconnection called a *bounds network* is adopted only for bounds data. The bounds network connects all four bounds nodes in a mesh style as depicted by gray lines in Figure 1(b) and allows communication between bounds nodes within two hops. 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. 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 1(b). A more detailed explanation can be found in [1].

IV. Performance Evaluation

To measure the performance of the proposed schemes, we use Simics full-system simulator [2] along with Full-system Execution-driven Simulator for x86 (FeS_2) [3], [4] to capture all memory reference traces. Then those traces are fed to a cycle-accurate CMP cache simulator that models different cache architectures: HardBound and BCache. HardBound [5] is the most recent mechanism that provides an architectural support for bounds checking. For parallel workloads, we use five PARSEC [6] benchmarks and our own parallel benchmark called *ParallelPointerBench* that generates more pointer-intensive workloads.

Figure 2 shows the performance of three different schemes in terms of micro-ops per cycle(μPC) for all memory operations executed; HardBound, BCache and BCache with skipping bounds checking. Using BCache and the smart tagging improves the performance by 29% on average compared to HardBound. 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 II. We can see that approximately 97% of bounds checking can be skipped on average.

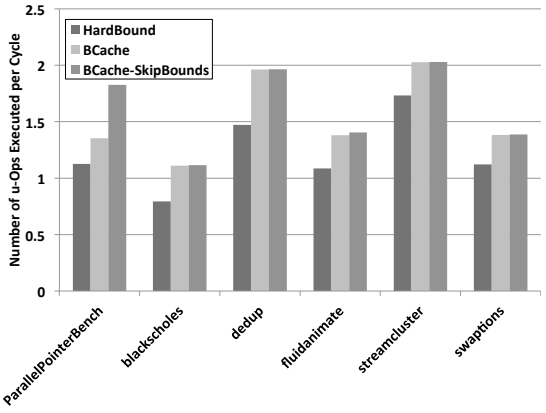


Figure 2: Performance improvement of memory accesses

Table II: 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%

V. Conclusions

We have proposed an architectural support for fast and efficient bounds checking for multi-threaded workloads in CMP environments. Simulation results show that the proposed scheme increases μPC of memory operations by 29% on average.

References

[1] B. An, K. Yum, and E. Kim, "Scalable and Efficient Bounds Checking for Large-scale CMP Environments," Texas A&M University CSE Dept., Tech. Rep. 2011-8-1, August 2011.

[2] 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.

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

[4] 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.

[5] 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.

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