SANDIA REPORT

SAND2017-0002
Unlimited Release
Printed January, 2017

Samba: A Detailed Memory Management Unit (MMU) for the SST Simulation Framework

A. Awad, S.D. Hammond, G.R. Voskuilen, and R.J. Hoekstra
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185

{aawad, sdhammo, grvosku, rjhoeks}@sandia.gov

Sandia National Laboratories

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2017-0002
Unlimited Release
Printed January, 2017

Samba: A Detailed Memory Management Unit (MMU) for the SST Simulation Framework

Acknowledgment

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

The Structural Simulation Toolkit is a high-performance, multi-node, threaded parallel discrete event simulation framework which supports the analysis of advanced computing architectures. We are grateful to the numerous funding sources who have contributed to the development of SST.

Contents

Chapter 1

Samba: A Detailed Memory Management Unit (MMU) for SST Simulator

1.1 Overview of Samba

The Memory Management Unit (MMU) is one of the most important parts of any modern computing system. It can be thought of as the hardware support for virtual memory, which enforces access permissions, and manages the translation of processes' virtual addresses into real memory physical addresses. The operating system (OS) allocates the physical memory in a small granularity frames that can be accessed by processes with appropriate permissions. Each process has the illusion of having an entire memory space, however, the actual addresses issued by the process, ('virtual addresses'), are translated into the physical frame address allocated by the OS through virtual memory.

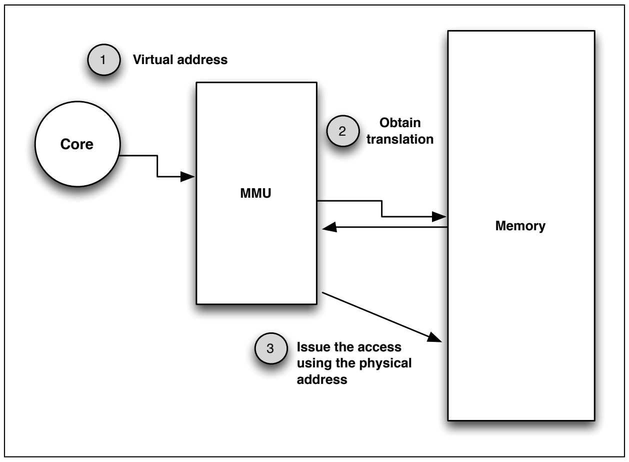

Figure 1.1. Simple Illustration of an MMU

Figure 1.1 depicts the role of an MMU in the system. Initially, a process running on a core issues a request using a *virtual address* to the MMU, which in turn checks if it has a cached translation of the virtual address. If the MMU does not have a cached version of translation, it will consult the page table, which is created by the OS. Once the translation is obtained and the permissions of access are checked, the obtained *physical address* will be used to forward the request to the memory sub-system.

Accessing the memory to obtain a translation for each memory request can be very expensive. For this reason, the MMU has internal caches to speed up the translation lookup process. The MMU caching structures are typically called *Translation-Lookaside Buffers (TLBs)* and *Page Table Walking Caches (PTWCs)*. To better understand the difference between TLBs and PTWC, Figure 1.2 shows the process of performing a page table walk to obtain the translation.

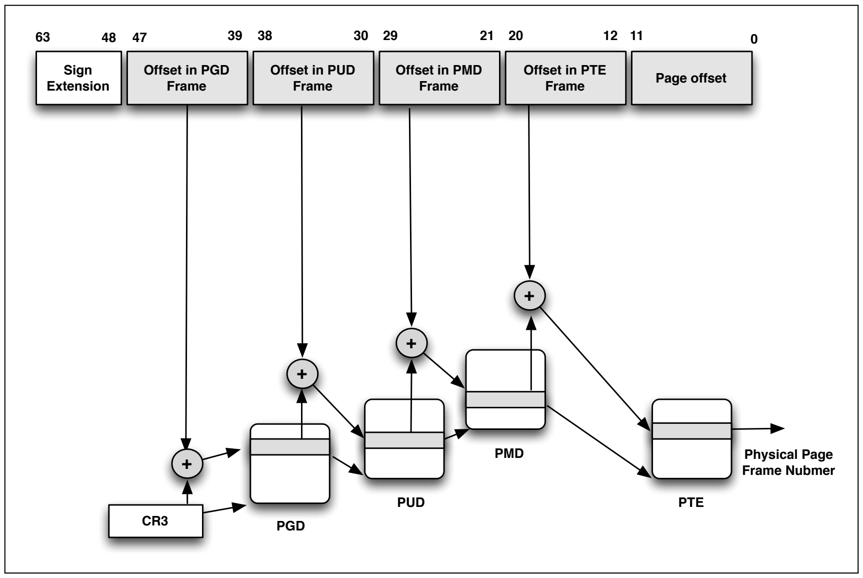

Figure 1.2. Illustration of the page table walk process.

Initially, if the translation is not found in the local cache of the MMU, a page table walk is triggered. The page table itself is organized as a radix tree of four levels (for x86-64 systems). The page table walking process starts with using the address of the root page of the page table, which is located in the CR3 register (for x86-64 systems). As shown in Figure 1.2, the page table walker derives the address of the next level through specific bits in the virtual address, to be used to index the current level. Finally, the leaf of the page table contains the translation (the corresponding physical address and the associated permissions). This process presents several candidates for caching. For instance, caching the translations from each level can help reducing the average number of requests to complete the translation process. As an example, if we can find the content of the PMD, we can use it to access the PTE frame, hence obtaining the translation with only a single memory access, instead of four.

PTWCs are the structures used to accelerate the page walking process through caching translations from different levels, while TLBs are used to cache the translations from the leaf of the page table, i.e, the actual translations.

Note that the page table walking process also depends on the page size, where using huge pages (2MB or 1GB) will require fewer page table walk steps to obtain the final translation.

1.1.1 Why is Modeling an MMU Important?

The performance of the MMU unit is critical, as it lies on the critical path of every memory subsystem access. Accordingly, MMU's structures, such as TLBs, are designed to be extremely fast and energy efficient, hence their small sizes. As mentioned earlier, the smallest granularity the OS deals with is physical frames with typical sizes of 4KB. Using 4KB pages brings flexibility in managing memory and exploiting the heterogeneity of pages' access pattern. This becomes more obvious in multi-socket systems and heterogeneous memory architectures. On the other hand, the use of 4KB pages can increase pressure on the MMU compared to using 2MB or 1GB pages.

The performance of the MMU may become a bottleneck for applications with poor temporal locality, *e.g.*, data analytics. A poor MMU design can incur at least one memory access to obtain the translation for each memory request. For huge memory footprint applications, where PTWCs are ineffective, each memory access can require up to four dependent memory accesses before proceeding. With the advent of potentially very dense emerging NVM technologies, such as 3D XPoint by Intel/Micron, and the continuously growing memory needs of applications, the design of the MMU is becoming considerably more challenging and critical to full system performance.

1.1.2 What Analyses can the Samba Model Support?

- **The Configuration of TLB Structures:** Samba enables varying the number of levels of TLBs, the size of each TLB structure, the number of entries of each page supported page size on each level, the associativity of each TLB structure, the maximum number of requests per cycle, the access latency and the maximum number of outstanding misses of each structure.
- **The Configuration of Page Table Walking Caches:** Samba enables varying the size of any PTWC, the associativity, the access latency and the maximum number of concurrent page table walking processes.
- **The Impact of using an Arbitrary Page Size on MMU Performance**.

The details of how to use Samba and vary its parameters will be discussed in the next sections.

Chapter 2

Samba Specification (Version 1.0)

In this chapter, we detail the specifications of the Samba SST element and model. First, Samba is implemented as an element of the Structural Simulation Toolkit (SST) simulator [2]. In the Samba source directory, there are four main classes:

- **TLB Class**: The TLB class is defined inside *TLBUnit.h* and implemented in *TLBUnit.cpp*. The TLB class is used to define TLB structures with the specified parameters. Each TLB instance models a single TLB hardware structure.
- **PageTableWalker Class**: The PageTableWalker class is defined inside *PageTableWalker.h* and implemented in *PageTableWalker.cpp*. The PageTableWalker class is used to define the Page Table Walker and the Page Table Walk Caching structures with the specified parameters. Note that the Page Table Walker class supports multiple concurrent requests, hence modeling parallel page table walkers.
- **TLBhierarchy Class**: The TLBhierarchy class is defined inside *TLBhierarchy.h* and implemented in *TLBhierarchy.cpp*. The TLBhierarchy class can be used to instantiate a private hierarchy of TLBs and Page Table Walkers. Each core or accelerator in the system will have its own private hierarchy. For instance, TLBhierarchy object can represent the MMU of an accelerator, IOMMU, or a private MMU of a specific core.
- **Samba Class**: The Samba class is defined inside *Samba.h* and implemented in *Samba.cpp* and *libSamba.cpp*. The Samba class is mainly used to instantiate TLB hierarchies in the simulation, however, a single Samba component can have multiple TLBhierarchy objects. It is the analyst's decision to have a Samba object for each core, or, instead, to have a single Samba object with multiple TLB hierarchies.

To put things together, Figure 2.1 depicts the architecture of Samba.

As mentioned earlier, a Samba object is the main container that either includes one TLBhierarchy object, similar to the one shown in Figure 2.1, or multiple TLBhierarchy objects. The main ports to the TLBhierarchy object are the following:

- **cpu_to_mmu**: This interface can be used to connect the MMU unit to a cpu core interface to the memory system, e.g., ariel core's *cache_link*, through a link.
- **mmu_to_cache**: This interface can be used to connect the MMU unit to the cache hierarchy, so it forwards the translated memory requests. As an example, connecting to the L1 cache's *high_network_0* interface, through a link.
- **ptw_to_mem**: This interface is used to connect the MMU's page table walker to the cache hierarchy or memory directly. Mainly used to obtain the translation through walking the page table. Depending on the design, this can be connected to the memory directly, tiny dummy caches then L3 cache, or to a self-link with fixed delay.

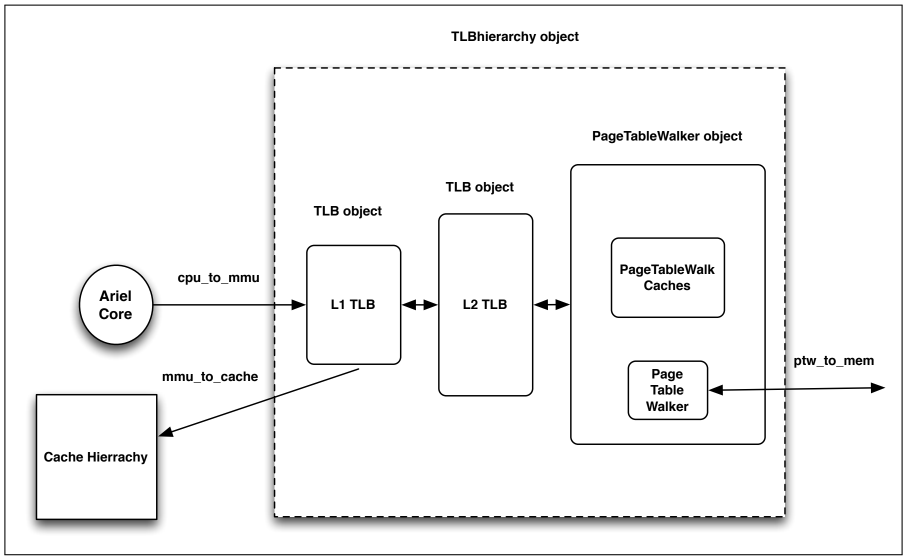

Figure 2.1. The architecture of Samba.

In the next section, we will study the details of each class and its parameters.

2.0.3 Examples of Samba Usage

In this section, we discuss the details of Samba units through examples. First, assume we want to build a two-level TLB hierarchy with the following specifications:

- TLB-L1: 32-entries for 4KB translation entries, with 4-way associativity. 8-entries for 2MB translation entries, with 8-way associativity.
- TLB-L2: 1024-entries for 4KB translation entries, with 16-way associativity. 256-entries for 2MB translation entries, with 16-way associativity.
- Page Table Walking Caches: For simplicity, we will model 4 caches with same configuration of 32-entries size and 4-way associativity, for each cache
- OS default page size: 4KB

Below is a code snippet to demonstrate the different parameters of each unit. Each parameter options has a comment describing it.

```
# Instantiating a samba instance
mmu = sst.Component("mmu0", "Samba")

# Adding the parameters for the Samba unit
mmu.addParams({
   "os_page_size": 4, # This represents the default OS allocated page size in KBs, you can choose
       whatever page size you want
   "corecount": 1, # This basically determines the number private TLB hierarchies for this Samba
       instance
```

```
  "levels": 2, # This tells Samba to instantiate to two levels of TLB, you can vary it if you
      want to examine deeper TLB hierarchies
  "clock": "2GHz", # This determines the clock frequency feeding this Samba unit
# L1 TLB
  "sizes_L1": 2, # This determines the number of page sizes supported of the translation entries
      in L1 TLB
  "page_size1_L1": 4, # This determines the first supported page size in the L1, in KBs
  "page_size2_L1": 2048, # This determines the second supported page size in L1, in KBs
  "size1_L1": 32, # This determines the number of entries supported for the first page size in L1
      TLB
  "size2_L1": 8, # This determines the number of entries supported for the second page size in L1
      TLB
  "assoc1_L1": 4, # This determines the associativity for the entries of the first page size in
      L1 TLB
  "assoc2_L1": 8, # This determines the associativity for the entries of the second page size in
      L1 TLB
  "latency_L1": 4, # This determines the latency (cycles) of accessing the L1 TLB structure
  "max_width_L1": 3, # This determines the number of requests can be processed per cycle in L1 TLB
  "max_outstanding_L1": 5, # This determines the maximum number of outstanding misses that can be
      handled in parallel for L1 TLB
  "parallel_mode_L1": 1, # This option is specific for L1 TLB, which allows L1 TLB hits to be
      serviced back in the same cycle, thus modeling a realistic parallel L1 TLB check and cache
      access
# L2 TLB
  "sizes_L2": 2, # This determines the number of page sizes supported of the translation entries
      in L2 TLB
  "page_size1_L2": 4, # This determines the first supported page size in the L2, in KBs
  "page_size2_L2": 2048, # This determines the second supported page size in L2, in KBs
  "size1_L2": 1024, # This determines the number of entries supported for the first page size in
      L2 TLB
  "size2_L2": 256, # This determines the number of entries supported for the second page size in
      L2 TLB
  "assoc1_L2": 16, # This determines the associativity for the entries of the first page size in
      L2 TLB
  "assoc2_L2": 16, # This determines the associativity for the entries of the second page size in
      L2 TLB
  "latency_L2": 10, # This determines the latency (cycles) of accessing the L2 TLB structure
  "max_width_L2": 2, # This determines the number of requests can be processed per cycle in L2
      TLB, note that there are no benefits from having this larger than the max outstanding
      misses of the previous level
  "max_outstanding_L2": 3, # This determines the maximum number of outstanding misses that can be
      handled in parallel for L2 TLB
# Page Table Walking Caches
  "size1_PTWC": 32, # The PTEs' cache size is 32 entries
  "assoc1_PTWC": 4, # The PTEs' cache associativity
  "size2_PTWC": 32, # The PMDs' cache size is 32 entries
  "assoc2_PTWC": 4, # The PMDs' cache associativity
  "size3_PTWC": 32, # The PUDs' cache size is 32 entries
  "assoc3_PTWC": 4, # The PUDs' cache associativity
  "size4_PTWC": 32, # The PGDs' cache size is 32 entries
  "assoc4_PTWC": 4, # The PGDs' cache associativity
  "latency_PTWC": 10, # This is the access latency (cycles) for accessing the page table walk
      caches in parallel
  "self_connected": 1, # This means that the page table walker will substitute fixed latency
      instead of actually accessing the memory to do the page walks
  "page_walk_latency": 200, # This is the fixed memory latency (in nanoseconds) for each page
      table walk memory access
# Note: the page_walk_latency is ignored if the page table walker is not self-connected (connected
    through ptw_to_mem to to the memory or cache hierarchy
  "max_outstanding_PTWC": 4, # this determines the number of concurrent page table walk requests,
      i.e., parallel page table walkers
  "max_width_PTWC": 2, # this represents the maximum number of requests can be processed per
      cycle, i.e, parallel checking of PTWCs
});
```

In the next section, we will study an example of the performance sensitivity to Samba's parameters.

Chapter 3

Application Performance Sensitivity to Samba Parameterization

In this chapter, we will vary different parameters for the Samba unit and study how this affects the performance.

3.1 Methodology

We run our simulation using the SST simulator [2] modified to integrate the Samba model. We use XS-Bench [3] miniapp as a case study for our demo. In our run, we simulate 8 cores with 2GHz frequency, while running 8 threads from XSBench for 100,000 Cross Section Lookups, with the small input set. The memory footprint generated by the run is roughly 227.5MB. For our cores, we model an issue width of 3 instructions and a maximum of 16 outstanding memory requests. Our default configurations for MMU is using 4KB translation with 32-entry and 4-way associativity L1 TLB, 1536-entry with 12-way associativity L2 TLB, and a PTWCs of 32-entry and 4-way associativity each. We also model the TLBs with a width of 3 requests per cycle and maximum outstanding misses of 2. The TLB access latency is 4 and 10 cycles for L1 and L2, respectively. The PTWC access latency is 10 cycles.

3.1.1 Page Size and L1 TLB Size

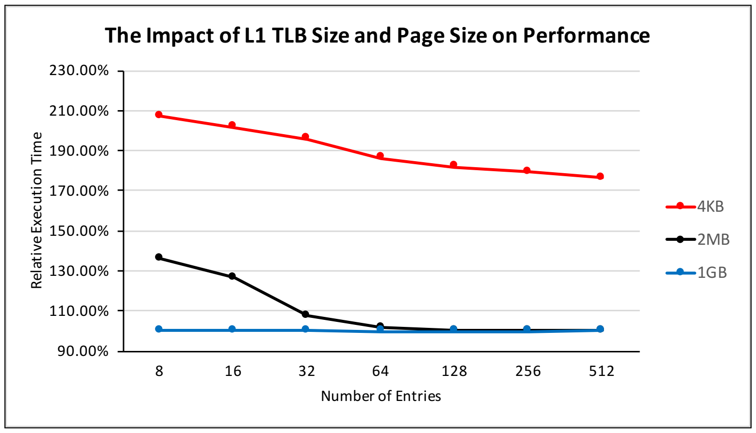

Figure 3.1. The impact of page size and L1 TLB size on performance.

As described above, Samba supports different page sizes and number of entries for each page size. Figure 3.1 shows the impact of performance of different page sizes while varying their corresponding number of entries.

We can observe that the L1 TLB size impact on performance differs on the page size used in the system. For instance, when using 4KB pages, the relative execution time goes from 206.9% at 8-entries down to 176.3% for 512-entries. In contrast, for 2MB pages, the relative execution time goes 136.2% at 8-entries to about 100% for 128-entries and larger. While using 1GB pages brings a performance overhead of about 0% (100% relative execution time), for any number of entries larger than or equal to 8.

Using large pages increases the memory reach. For instance, using 8-entries of 4KB pages only covers translations for 32KB memory footprint. In contrast, a 2MB 8-entries L1 TLB covers a 16MB memory footprint. Using 1GB 8-entries L1 TLB is sufficient to cover 8GB of the memory footprint. For our run, the memory footprint is 227.5MB, which explains why using 1GB pages makes the MMU overhead negligible; about 100% L1 TLB hit rate. For the 2MB pages, as we would expect, 128-entries is sufficient to cover the whole memory footprint (227.5MB) of our run, hence minimizing the MMU overhead. Finally, a 4KB pages poorly benefits from increasing the number of entries, due to the inability to cover large percentage of the memory footprint.

While using 1GB or 2MB pages may look appealing and a straight forward solution, this comes at the cost of significant performance overhead when dealing with multi-socket system or heterogeneous memory architecture in general [1]. In summary, using large pages limits allocating physical memory based on the access proximity. The trade-offs of using large pages are worth further study that is beyond the scope of this report, however, Samba enables such kind of studies.

3.2 The Impact of Parallel Miss Handling Capacity of L1 TLB

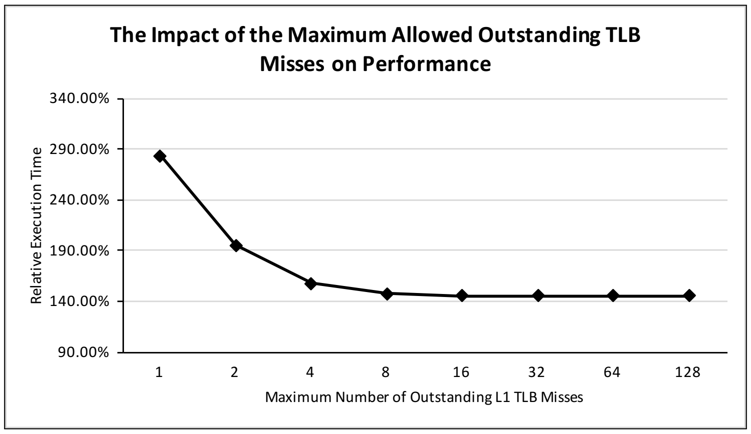

Figure 3.2. The impact of the capacity of handling concurrent TLB misses on performance.

Modern processor systems allow multiple outstanding memory requests to be handled in parallel. Accordingly, Samba models handling concurrent misses at each level of its Units. Figure 3.2 shows the performance sensitivity of varying the number of the allowed outstanding misses at the L1 TLB. We change this value for all units in the MMU hierarchy to avoid being throttled by lower level units.

We can observe that increasing the number of misses can be handled in parallel has significant impact on performance. However, there is a point where a larger window of parallel misses handling does not provide any performance gain. Specifically, in the figure, we can see that beyond 16 parallel misses handling, there are no additional performance gains. The reason behind this is that our core model can handle a maximum of 16 memory requests in parallel before stalling.

Note that allowing a large number of parallel misses complicates the MMU design and affects the power budget negatively. Accordingly, Samba enables studying the performance trade-offs for different numbers of allowed concurrent misses, and if it is worth it to design larger windows.

3.2.1 The Impact of Page Table Walking Caches (PTWCs) on Performance

PTWCs allow skipping steps from the page table walking process through caching entries form different levels of the page table. To study its impact on performance, we vary the size of every PTWC from 8 up to 1024. Figure 3.3 shows the performance sensitivity due to using larger PTWC caches.

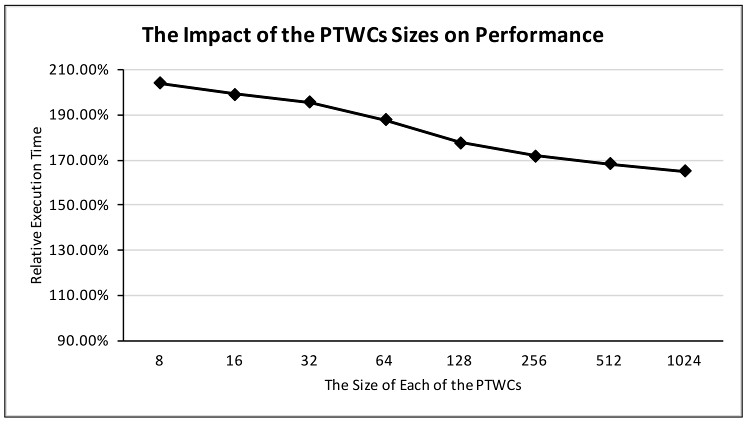

Figure 3.3. The impact of the capacity of the PTWCs on performance.

We can observe that increasing the PTWC can improve the performance, but with similar scalability to increasing the L1 TLB size. The performance improves from 203.8% relative execution at 8-entries down to 164.8% at 1024-entries.

Chapter 4

Conclusion

In conclusion, we have presented Samba as a detailed Memory Management Unit (MMU) model for SST simulation framework. We explained the options and paramaters can be configured and how would this affect the performance. Finally, we used the XSBench mini-app as a short case study to show how application performance can vary with differing configurations of Samba's components. We hope that Samba will be used to evaluate the impact of MMU on future system for different workoads.

References

[1] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quema. Large Pages May Be Harmful on NUMA Systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 231–242, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/gaud`.

[2] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, R Risen, Jeanine Cook, Paul Rosenfeld, E CooperBalls, et al. The Structural Simulation Toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[3] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto.

Sandia National Laboratories