NTNU
Norwegian University of Science and Technology

Designing a Virtual Memory System for the SHMAC Research Infrastructure

Audun Sutterud

Master of Science in Computer Science
Submission date: August 2017
Supervisor: Magnus Jahre, IDI

Norwegian University of Science and Technology
Department of Computer Science

Dedicated to Benedikte Bøe, for her patience and love.

Abstract

The Single-ISA Heterogeneous MAny-core Computer (SHMAC) is an infrastructure for realizing heterogeneous computing systems. The current SHMAC prototype does not have a Memory Management Unit (MMU). An MMU would simplify the process of providing a process abstraction on the SHMAC and make it possible to run multiple programs at the same time.

This master thesis provides a qualitative comparison of different techniques and solutions for implementing virtual memory. These are discussed with regard to the SHMAC architecture and recommendations are made as to how virtual memory should be implemented in the SHMAC. Specifically, three different designs are identified by locating the address translation hardware either before the L1 cache, after the L1 cache, or before a possibly distributed L2 cache. All three designs seem to be viable alternatives but further investigation is necessary to determine which design is the best fit for SHMAC.

Sammendrag

Single-ISA Heterogeneous MAny-core Computer (SHMAC) er et rammeverk for å realisere heterogene datasystemer. Den nåværende SHMAC-prototypen har ingen minnebehandlingsenhet. En minnebehandlingsenhet vil forenkle prosessen med å implementere prosessabstraksjonen i SHMAC og gjøre det mulig å kjøre flere programmer samtidig.

Masteroppgaven gjør en kvalitativ sammenligning av forskjellige teknikker og løsninger for implementasjon av virtuelt minne. Disse diskuteres i henhold til SHMAC-arkitekturen og det gis anbefalinger om hvordan virtuelt minne bør implementeres i SHMAC. Mer konkret, tre ulike design identifiseres ved å plassere hardware for addresseoversettelse enten før L1 cache, etter L1 cache, eller før en muligens distribuert L2 cache. Alle tre designene virker som brukbare alternativer, men videre undersøkelse er nødvendig for å avgjøre hvilket design som passer best i SHMAC.

Acknowledgment

First I wish to thank Associate Professor Jahre for our technical discussions and for his practical advice. I also wish to thank Higher Executive Officer Drøsshaug for guiding me through the practical details of writing and delivering a masters thesis and Senior Adviser Tangen for our useful discussions work structure and self motivation. I am grateful to my friends and fellow students for lending me their companionship and sharing with me their similar writing experiences. My family also deserves many thanks for their unfailing support and especially their persistent encouragement towards the end. Finally, my deepest gratitude goes to Benedikte Bøe for her kind patience and love. Her presence has been a source of inspiration and motivation throughout all my work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since 1965, Moore's law has correctly predicted that the number of transistors in a microprocessor chip doubles every second year [1][2]. Using Dennard scaling to scale the voltage proportionally to transistor dimensions it has been possible to utilize all transistors while avoiding problems with heat dissipation [3]. The increased transistor density has made it possible to exploit ILP by using more complex microarchitectures, hiding the memory latency by adding on-chip caches, and improving throughput by adding multiple homogeneous on-chip cores, enabling a more than 1000-fold improvement in microprocessor performance over the past 25 years.

The breakdown of Dennard scaling in recent years poses a problem. While transistor dimensions are still scaled down, static power leakage at low voltages makes further voltage reduction an ineffective measure to manage heat dissipation. As a result, to stay within the power budget not all transistors can be active at the same time, leading to unused area termed as *dark silicon* [4]. This has motivated research into heterogeneous computing in which multicore processors are constructed from computation units with different processing characteristics. Given a set of tasks, the processor dynamically allocates tasks to the most suited units and keeps the remaining units inactive to stay within the power budget. Increased transistor density will enable increased customization, increasing performance without relying on Dennard scaling [5].

The Single-ISA Heterogenous MAny-core Computer (SHMAC) is a tile-based processor architecture for heterogeneous computing research developed as part of a research project by the Energy Efficient Computing Systems group at the Norwegian University of Science and Technology [6]. It is a heterogeneous processor that organizes the computational units using a rectangular grid of tiles. Each tile is locally connected to each of its four neighbors and instructions and data are propagated through the grid. The currently supported tiles include among others: a processor tile executing the RISC-V instruction set architecture (ISA); a main memory tile, which contains an L2 cache and communicates with off-chip memory; and several accelerator tiles with different processing characteristics. The RISC-V ISA is an open source RISC ISA designed to support computer architecture research [7][8].

It is desired to run multiple programs on the SHMAC at the same time. However, SHMAC currently runs a bare-metal environment in which only a single program can be executed. The ideal solution would be to port an operating system to SHMAC in which multiple programs would run as processes. Most modern operating systems rely on virtual memory to implement processes, a technique used to share limited memory resources between multiple programs. While many architectures provide hardware support to simplify the implementation of virtual memory, no such hardware support is present in the SHMAC. To simplify the porting of operating systems, SHMAC should provide some form of hardware support for virtual memory.

This master thesis provides a qualitative comparison of different techniques and solutions for implementing virtual memory. These are discussed with regard to the SHMAC architecture and recommendations are made as to how virtual memory should be implemented in the SHMAC. Emphasis is placed on performance overhead, hardware area, and the effort necessary to port operating systems onto the platform. Specifically, three different designs are identified by locating the address translation hardware either before the L1 cache, after the L1 cache, or before a possibly distributed L2 cache.

Chapter 2

Background

2.1 Virtual Memory Overview

Virtual memory is a technique used to share limited memory resources between multiple programs. Most modern operating systems rely on virtual memory to implement processes.

2.1.1 Address Spaces

With virtual memory, each process owns a set of *virtual addresses* residing in a *virtual address space.* Access to physical memory is provided by translating virtual addresses to physical addresses on the fly. Processes are isolated from each other by restricting access to virtual addresses owned by other processes, and memory is shared by translating virtual addresses of different processes to the same physical addresses. To free up memory, rarely used data can be placed in secondary storage.

Virtual addresses are grouped into contiguous blocks called *virtual pages*, each identified by a virtual page number (VPN), and physical addresses are grouped into *page frames*, each identified by a page frame number (PFN). A partial mapping is maintained from a virtual address space to a physical address space by mapping VPNs to PFNs.

To enable sharing, some virtual memory systems allow multiple virtual addresses to map to the same physical address. Such mappings are called *synonyms.*

2.1.2 Page Tables

Translation information for a virtual address space is stored in a data structure called a *page table.* The page table is indexed by VPNs to access PTEs, each which contains the translation information associated with a single virtual page. A PTE can include information such as the PFN in memory, the location of the page in secondary storage, and protection information used to enforce access restrictions such as read-write and read-only permissions. A more comprehensive list is given in Table 2.1. The operating system maintains the page table by updating the PTEs:

page frame number	used to translate the virtual addresses of the page
present bit	indicates that the page is present in memory
page size	the size of the page
disk location	the disk location of the page
protection information	used to enforce access restrictions
address space identifier	identifies the process that owns the page
dirty bit	indicates that the page is modified
use bit	indicates that the page has been accessed

Table 2.1: Information that can be contained in a PTE.

it can map virtual pages to page frames, demap virtual pages from page frames, and modify page-level protection information (*rights change*).

There are two approaches: either, each process has its own page table (*private address spaces*), or a single page table is shared among all processes (*global address space*). Private address spaces are usually kept disjoint by extending the virtual addresses with a prefix unique to the address spaces, such as address space identifiers (ASIDs) or *segments* [9]. A global address space must be large to accommodate the address needs of all processes and therefore it is better suited for systems that support wide addresses such as 64-bit architectures. A global address space can also be used in 32-bit architectures by extending the virtual addresses on reference, either through ASIDs or segments. An advantage of using a global page table is that it considerably reduces management overhead associated with managing a page table for each process [10].

2.1.3 Page Faults

When a hardware event occurs that needs to be handled in software, the processor is interrupted by a *page fault* to execute a page fault handler set by the operating system. A page fault is typically generated when a non-present page is accessed. The accessed page is then mapped to an unused page frame and if necessary loaded from disk (*paged in*). If there are no unused page frames, another page is written to disk (*paged out*) to make one available, a process known as *page eviction.* To reduce the size of the page table, pages can be left unmapped until they are first accessed. Page faults are then generated when unmapped pages are accessed to create the necessary mappings.

2.1.4 The Translation Lookaside Buffer

The virtual addresses issued by the processor are translated to physical addresses at some location along the path to physical memory according to the translation information stored in the page table. To reduce the number of page table accesses, translation information is often buffered locally in a special cache called a translation lookaside buffer (TLB). On a memory access, if the translation information is present in the TLB (*TLB hit*), the memory access proceeds with little or no overhead; otherwise (*TLB miss*), the required information must be loaded from the page table into the TLB before the memory access can proceed. TLBs in modern architectures typically contains from 16 to 512 entries with each entry buffering information for one or two PTEs [11]. The amount of memory accessible from the TLB is called the *TLB reach* and is calculated as the number of entries multiplied by the page size. [12].

The location of the TLB directly affects the organization of the caches in the system by determining if the bits used as index and tag originate from the physical or the virtual address. Three different organizations are considered in this article: physically indexed, physically tagged (PIPT); VIPT; and VIVT. A cache in which index bits originate from the virtual address is called *virtual*, other caches are called *physical*. Because a virtual cache implicitly stores translation information, in some cases they can be accessed without accessing the TLB at all.

TLBs and caches that store translation information can become inconsistent with respect to the page table. When multiple TLBs are used, different mappings can exist for a single virtual address (*homonyms*). When synonyms are used, there can be circumstances under which synonyms are created inadvertently (*aliases*). When access rights are buffered in a virtual cache, the buffered access rights can become invalid (*stale access rights*). To maintain correct operation, the virtual memory system must handle homonyms, aliases and stale access rights and ensure that they are never used.

2.1.5 Hardware Support for Simple Optimizations

Specific hardware support is necessary to enable some performance optimizations common in virtual memory systems. To reduce the overhead of page eviction, each page can be associated with a *dirty bit* that is set when the page is modified. Clean pages can then be evicted without having to access the disk. The operating system can maintain a pool of clean pages by writing dirty pages to disk when the disk is idle.

To reduce the frequency of page faults, a page eviction algorithm such as least recently used (LRU) can be used to make better decisions as to what page should be evicted. With LRU, a *use bit* is associated with each page and a usage history is obtained for every page by periodically reading and clearing use bits. When evicting a page, the least recently used page is selected on the assumption that other pages are more likely to be used in the near future.

2.2 The SHMAC Architecture

The SHMAC [6] is a tile-based architecture with a mesh interconnect [13]. Computational units are organized as a rectangular grid of tiles where each tile is locally connected to its four closest neighbors; north, south, east and west. Connections do not wrap around edges. SHMAC-architectures are realized in in FPGA and the tiles are configured at synthesis time. An overview of the architecture is given in Figure 2.1. SHMAC supports a number of tiles:

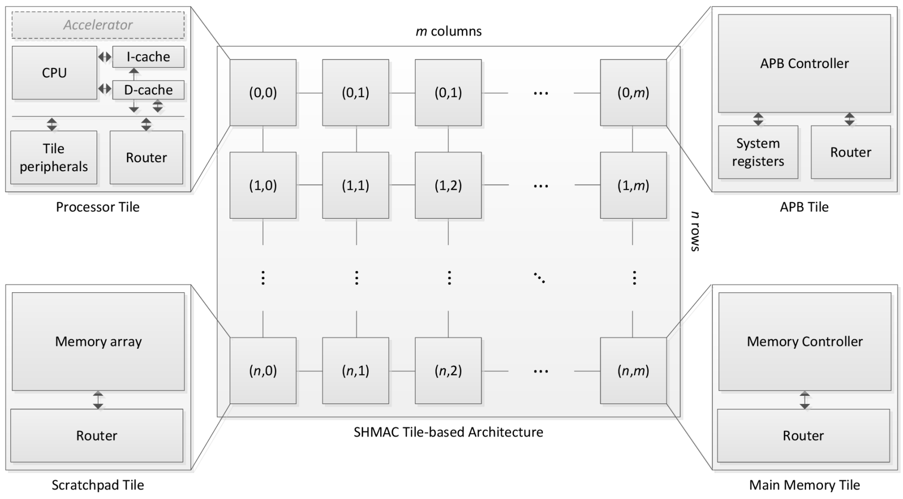

Figure 2.1: The high-level architecture of SHMAC.

- a general-purpose processor tile;
- a main memory tile that communicates with off-chip memory;
- an Advanced Peripheral Bus tile to communicate with the FPGA system;
- an accelerator tile that performs specialized computation such as SHA-1;
- a scratchpad tile that implements a scratchpad memory; and
- a dummy tile, used to fill unused tile slots.

The accelerator and dummy tiles can also be integrated into a processor tile.

2.2.1 Processor Tiles

A processor tile consist of a processing core, separate instruction and data-caches, a router, a local scratchpad and tile registers. It can be configured to contain a global scratchpad and an accelerator, but these are optional.

The processor tile executes the complete 32-bit RISCV base integer instruction set plus standard extension A for atomic instructions [14]. Some instructions can be left unimplemented to conserve FPGA resources; when these instructions are executed, the processor is interrupted and the instruction is emulated in software.

2.2.2 Caches and the Memory Hierarchy

The memory hierarchy consists of local L1 caches located at each processor tile, a global L2 cache, and an external main memory. All caches are implemented by a single, configurable cache module. A cache is accessed in two stages: the first stage reads data from the cache memory and the second stage handles the request depending on what was read. This results in a two cycle latency for cache reads. Writes can complete in a single cycle if the cache is configured to support non-blocking writes [15]. Caches also have some optional components. A miss status holding register can be used to better serve simultaneous read requests. A write buffer can be used to buffer writes when the interconnect is busy. Also, a merging write buffer can be used to merge different writes.

Cache coherency must be maintained between L1 caches at different processor tiles. Three different techniques are currently used to address this issue:

1. placing shared data in uncacheable regions;
2. turning off the cache when accessing shared memory; and
3. flushing or invalidating the cache.

All three techniques can be expected to exhibit inferior performance when compared to using a more sophisticated cache coherency protocol that allows accessing shared data directly in the cache. At the time of this writing, a theses project to implement a more efficient cache coherency protocol in hardware is being worked on [15].

Chapter 3

Virtual Memory Design

This section discusses design choices that must be considered when designing a virtual memory system. The discussion is simplifying by focusing on architectures in which there are two levels of cache in the memory hierarchy: a private L1 cache located at each core, and a single unified L2-cache located between the L1 caches and the main memory. Note that this is different from the SHMAC memory system in which the L2-cache can be distributed around the grid (see Section 2.2).

3.1 Cache Organization

This section discusses the placement of the TLB and the organization of the L1 caches. The cache organization depends on the TLB location and has important consequences for the characteristics of the overall system. Table 3.1 summarizes the characteristics of each alternative with respect to implementation and performance.

3.1.1 TLB Before a PIPT L1 Cache

The TLB can be located before a PIPT L1 cache. Access to the cache can either be pipelined or parallel.

When the access is pipelined, the TLB is placed on the critical path of every access to the L1 cache. This can make it problematic to support single-cycle accesses to the L1 cache at high frequencies. For this reason, modern architectures seldom opt for this approach [9]. No translation information is buffered in the cache and therefore translation buffer consistency can be maintained simply by flushing the relevant entries from the TLB whenever the page table is updated.

When the access is parallel, the index bit used to access the cache must come from the page offset bits from the virtual address. These bits are not translated by the TLB and can be used to index the cache immediately without waiting for translation. It is still necessary to translate the tag bits and therefore the TLB access latency is only partially hidden. A disadvantage of this approach is that the restriction on the index bits effectively limits the cache size to the size of a page times the number of ways [16]. The cache size can still be increased by increasing the number of ways but this also increases the hardware complexity and latency of tag comparison [17].

3.1.2 TLB Before a VIPT L1 Cache

The TLB can be located before a VIPT L1 cache. This approach is similar to the previous one but by using virtual bits to index the cache it lifts the restriction on cache size. It also gives the compiler better control over where data is placed in the cache [18], allowing optimization techniques such as *blocking* [19]. A disadvantage of using virtual bits to index the cache is that the cache is vulnerable to aliases (see Section 3.7). The tag is still physical and therefore aliases can only occur in different sets. Cache coherence protocols that operate on physical addresses may require hardware to perform reverse translation in order to invalidate elements in the L1 cache.

3.1.3 TLB After a VIVT L1 Cache

The TLB can be located after a VIVT L1 cache.

An advantage of using virtual tags is that the TLB can be removed from the critical path. Following it can be large and slow without affecting the access time of the L1 cache, increasing the TLB reach and reducing the frequency of TLB misses. Also, there has been a trend toward computing in memory and communication interfaces. Moving the TLB down the memory hierarchy can make virtual addresses available where they are needed.

There are some issues when moving the TLB down the memory hierarchy. Cache access must still be checked against protection bits for process isolation and dirty bits and use bits must still be updated for performance. A common solution is to buffer information in the cache, even though this increases the size of the PTEs and the total cache area [11]. By relaxing the requirements for use and dirty bits they can be removed from the TLB entirely [20]. Use bits can be replaced by *miss bits* that are stored in the page table and set only when an access to a cache block misses. Accesses to present cache blocks does not set the miss bit, thus a page can be paged out even if it was used very recently. A similar approach can be used for dirty bits by setting the dirty bit whenever a writable block is brought into cache, with the disadvantage that dirty bits may be set even if the page has not been modified, causing unnecessary disk activity.

There are some disadvantages. With wide addresses, more bits are needed for the tags in a VIVT cache than in a VIPT cache. For example, with 64-bit virtual addresses, 36-bit physical addresses and 32 byte cache lines, a VIPT cache is about 10% larger [18].

3.1.4 TLB Before the L2 Cache

As bus addresses are virtual, no reverse translation is necessary for cache coherency protocols.

Design Choice	Implementation Issues	Performance Issues
PIPT (pipelined)	None	TLB access is fully on the critical path of cache access
PIPT (parallel)	L1 index size is restricted by page size	TLB access is partly on the critical path of cache access
VIPT	Aliases can occur in different sets, coherency protocol may require reverse translation	TLB access is partly on the critical path of cache access, the compiler can control data placement
VIVT	Aliases can occur, coherency protocol may require reverse translation, the cache buffers translation information, computing in memory is supported	A large TLB can be located further down the memory hierarchy, the compiler can control data placement

Table 3.1: Comparison of cache organizations.

3.2 Page Table Organization

The organization of the page table impacts both its storage size and access latency, usually with a trade-off between the two. The two main categories of page tables are *hierarchical* and *inverted*.

3.2.1 Linear Page Table

The PTEs can be stored in a linear structure to get excellent access time. By using the VPN as an offset into the page table, only a single memory reference is needed per access. Unfortunately, it also has a large memory footprint: with 32-bit virtual addresses, a page size of 4 KB, and a PTE size of 4 bytes, the table itself occupies 4 MB of physical memory. Allocating this amount of memory for the page table of every process is usually not acceptable. Also, for larger address spaces the problem becomes worse [10].

3.2.2 Hierarchical Page Table

Hierarchical page tables use a *root page table* to map pages containing *user page tables*. A user page table either maps pages containing another level of user page tables, or maps the pages used by the process. The root page table is typically pinned down, while the user page tables are left unmapped to be paged in on need, allowing the hierarchical page table to occupy much less space in memory than the linear structure. A disadvantage is that table access requires an additional memory reference for each level below the root. One optimization to this is to check the bottom layer directly for a valid page, and use the top-down table-walk as a fallback strategy. However, for large address spaces the access latency can still become a limiting factor [10].

Organization	Performance Issues
Linear	One memory reference per access, memory footprint is proportional to the number of pages in the virtual address space
Hierarchical	Memory references per access increases with the number of levels, memory footprint is proportional to the number of pages accessed by the process
Inverted	Memory references per access depends on the average length of the collision chains, memory footprint depends on the number of page frames

Table 3.2: Comparison of page table organizations.

3.2.3 Inverted Page Table

An inverted page table uses a different approach in that it contains a PTE for every page frame. Virtual pages are mapped onto page frames by hashing the VPN and using the hash to select a PTE. Because the number of page frames usually is much smaller than the number of pages, inverted page tables can be more space efficient than hierarchical page tables, especially for large address spaces. In the simplest organization, the hash is used to index a linear structure of PTEs and collisions are handled by an internal collision chain. On average, a table access requires a number of references equal to the average length of the collision chains. Increasing the size of the linear structure will keep the collision chains short, but increases the size of the table. PTEs are also larger than in a hierarchical page table because they must contain a PFN and a pointer to the next PTE in the collision chain, amplifying the effect [10].

3.3 TLB Management

Mapping information can be loaded into the TLB either by hardware or software. The hardware approach uses a finite state machine to access the page table directly in memory, a process known as *page table walking.* The hardware-based scheme is efficient, but it places constraints on the organization of the page table and makes it harder to port operating systems onto the hardware. The software approach generates a page fault to have the page fault handler load the information explicitly. The hardware never accesses the page table directly, thus no constraints are placed on the page table organization and porting efforts are smaller. However, the handler code can be 10 to 100 instructions long, taking many cycles to complete. A cache miss occurs if the handler code is not present in the instruction cache, and in pipelined cores the pipeline is flushed [9].

TLB Management	Implementation Issues	Performance Issues
Hardware	Page table organization is restricted	Efficient
Software	Page table organization is not restricted	Potentially inefficient

Table 3.3: Comparison of TLB management schemes.

3.4 Shared Memory

Shared memory increases memory utilization and efficiency by allowing processes to share program code or dynamic libraries and is used extensively in modern operating systems. In order to support efficient implementation of shared memory the virtual memory system must provide hardware support. This section discusses different ways shared memory can be supported in virtual memory systems.

3.4.1 Virtual Address Aliasing

With virtual address aliasing, multiple virtual pages are mapped to a single page frame. When the virtual pages belong to different processes, these processes share the physical page. Depending on the implementation of access rights, the processes may have different access rights to the physical page. The disadvantages of this approach is the additional overhead of maintaining multiple mappings. When a physical page is relocated, all virtual pages mapped to it must be updated. The multiple mappings can also compete for space in the TLB [10].

3.4.2 Shared Portions of Page Tables

An alternative that reduces the impact of managing multiple mappings is to share larger portions of the page table instead of sharing individual mappings. Such an approach can easily be implemented in a hierarchical page table. The disadvantages of this approach is that it increases the granularity with which memory can be shared and that it makes it harder to give different processes different access rights to the shared memory [10].

3.4.3 Global Address Space

A global address space can support sharing at small granularity and little overhead. Either the architecture must support wide addresses or addresses must be extended on reference. Modern operating systems may have poor support for global address spaces and therefore the efforts of porting them may be increased [10]. special hardware is needed to support different access rights (see Sectionsec:process-isolation).

3.5 Process Isolation

The task of maintaining process isolation is typically delegated to the virtual memory system for efficiency reasons. There are several ways to do this, depending on

Sharing Technique	Implementation Issues	Performance Issues
Virtual address aliasing	Used with private address spaces	Allows sharing at the granularity of a page, additional overhead of maintaining multiple mappings
Shared portions of page tables	Used with private address spaces	Allows sharing at the granularity of a portion of the page table
Global address space	Require extensive porting efforts, special hardware is needed to supporting different access rights	Allows sharing at a small granularity

Table 3.4: Comparison of techniques to implement shared memory.

whether private or global page tables are used. This section compares several solutions.

When private address spaces are used, processes can be associated with an address space by associating them with the ASIDs. When a global address space is used, a process can reference all possible virtual addresses. Isolation is achieved by executing processes in different *protection domains* that specifies access permissions for the different pages.

3.5.1 Flush on context switch (private address spaces)

The simplest way to maintain process isolation is to flush the TLB and any virtual caches on a context switch. The first access to any cache block generates a miss and is verified directly against the page table.

Flushing the TLB increases the frequency of TLB misses. Flushing the virtual cache also increases the frequency of cache misses, but this effect can be negligible in small caches with short warm-up time [21]. In caches that use a write-back policy, a flush can cause a considerable number of write-backs, increasing the context-switch latency. It may be possible to distribute such writes in time [21].

3.5.2 Tag translation information with ASID (private address spaces)

Flushes on context switch can be avoided by using ASIDs [22]. This approach has been used in several architectures [9]. The entries in the TLB and any virtual caches are extended to hold the ASID of the address space to which the translation information in the entry belongs, and a protected register is used to hold the ASID of the current process. On memory access, the ASID of the process is matched against the ASIDs in the TLB and the virtual caches.

When ASIDs are used, flushing is not necessary on context switch unless there are globally shared pages [10]. However, flushing is still necessary when the operating system recycles old ASIDs during process creation. The frequency of ASID recycling is affected by the number of unique ASIDs supported by hardware. Selective flushing by ASID can also be supported.

3.5.3 The domain-page model (global address space)

The domain-page model associates access permissions with each pair (domain, page) [18]. A register is used to hold the protection domain identifier (PD-ID) of the running process, and a protection lookaside buffer (PLB) is used to buffer access permissions associated with domain-page pairs. On memory access, the PLB is accessed with the PD-ID of the process and the VPN of the accessed page to retrieve access permissions. On a PLB miss, the permissions must be fetched from the page table before the access proceeds. If the process does not have permissions to access the page, an exception is generated regardless of whether the access hits cache or not.

The PLB is indexed using virtual addresses and can be accessed in parallel with the cache. On mapping changes, the PLB must be purged.

One might consider using the PLB as a TLB to hold both translation information and permissions. However, translation information is then duplicated for different domain-page pairs in which the page is identical, leading to unnecessary area overhead. Instead, moving access permissions from the TLB to the PLB makes it easier to locate the TLB after the L1 cache (see Section 3.1). Because the model separates translation and protection, it also allows optimizations in granularity and protection.

3.5.4 The page-group model (global page table)

The page-group model groups pages into *page-groups*; permissions are associated with each page-group, and one or more page-groups are associated with a domain [18]. This was implemented in the Hewlett-Packard PA-RISC architecture [23]. A set of registers holds *page-group identifiers* that identify the page-groups accessible to the protection domain of the running process. Each TLB entry includes a page-group identifier and page-group access permissions along with translation information. On memory access, the page-group identifier retrieved from the TLB is matched against the page-group identifiers in the registers. On a match, the access type is verified against the access permissions. On insufficient permissions or failure to match the page-group identifier, the processor can be interrupted to pass control to the operating system. The operating system may either modify the TLB and page-group registers and restart the instruction, or it may generate a protection fault.

As the TLB is accessed on every reference, this approach does not allow the TLB to be located after the L1 cache.

3.6 Maintaining TLB Consistency

Page-mapping changes can cause translation information buffered in the TLB to become inconsistent with respect to the page table. To maintain correct operation, the virtual memory system must ensure that inconsistent or stale translation information is never used.

Technique	Requirements	Performance Considerations
Flush on context switch	Private page tables	TLB and virtual caches are flushed on context switch
Tag translation information with ASID	Private page tables, register to buffer ASID, entries in the TLB and cache are extended with an ASID field	TLB and virtual caches are flushed on ASID recycle
Domain-page model	Global page tables, register to buffer PD-ID, PLB to buffer access permissions	Mapping changes cause PLB purge
Page-group model	Global page tables, registers to hold page-group identifiers, TLB entries can hold page-group identifiers and permissions, the TLB is located on the critical path	Mapping changes cause page-group identifiers to be purged

Table 3.5: Comparison of techniques to enforce process isolation.

In single-core systems, a simple solution is to flush translation information from the TLB as the page table is updated. In multi-core systems, however, a more sophisticated solution is required. Different cores can access the same translation information, making it necessary to involve multiple TLBs in the consistency actions. To see why different cores can access the same translation information, observe that a single page can be shared between multiple processes and that a single process can be scheduled on more than one core [24]. It is also necessary to eliminate race conditions that result from buffer hardware accessing the page table during update [25].

Many different solutions exist. They differ in characteristics such as the processor execution and idle time that result from maintaining tlb consistency, but also in implicit side-effects such as serialized page-table modifications, increased swapping activity, an increased number of TLB misses, and the inability to use time-saving optimizations [26]. Some solutions solve the problem by flushing stale translation information at the time of update, others instead use some strategy to prevent stale information from being accessed.

3.6.1 Typical hardware support

There are often special instructions that the processor cores can use to flush their TLBs. There can be an instruction to completely flush all translation information, but instructions can also exist for selectively flushing only parts of the TLB. Some types of partial flushing include flushing the mapping for a virtual address, flushing all synonyms for a physical address, or flushing all mappings given an ASID.

3.6.2 TLB Shootdown

TLB shootdown is a common algorithm for managing TLB consistency in software with minimal hardware support. In the classical TLB shootdown algorithm, processing cores are interrupted to perform consistency actions on their TLBs. The algorithm proceeds in two phases: first, the core that updates the page table (*initiator*) sends an interrupt to other cores (*responders*) if the operation might result in an inconsistency; second, the responders receive interrupts and carry out inconsistency actions. The operating system approximates the set of TLBs caching a mapping so that the initiator does not have to interrupt all cores [27].

The minimal hardware requirements for this solution is a set of interrupt lines that can interrupt the processors, and a mechanism for flushing TLB entries such as a special processor instruction. The cost of a single TLB shootdown increases with the number of processors. The frequency of TLB shootdowns is mainly workload dependent; workloads that perform memory or file I/O causes more page-remappings and triggers more TLB shootdowns. For the Linux 2.6.36, the set of responders is a relatively poor approximation, including most cores at every shootdown with about 80% being false positives on average. The percentage of compute cycles lost on executing TLB shootdowns seems to increase linearly with the number of cores. For certain workloads, it can approach as much as 25% for 64 cores, or 50% for 128 cores [27].

3.7 The Synonym Problem

The presence of synonyms can complicate the management of virtual caches by allowing multiple copies of the same data to exist at different locations in the cache. Writing to a copy causes the other copies to become stale and creates a consistency problem.

First note that there is no synonym problem in PIPT caches: the synonyms of a physical address all map to the same set and copies are avoided implicitly through tag comparison. Virtual caches are different in that they are indexed with virtual address bits. Synonyms with different index bits map to different sets, therefore tag comparison is no longer sufficient to detect copies. The possible sets that the synonyms of a physical address can map to form the *superset*. The size of the superset can be computed as two to the power of the number of virtual index bits. Recall that page offset bits are not translated, thus those index bits are not counted.

VIPT caches can exploit the fact that copies share the same tag. In VIVT caches, also the tag is virtual.

3.7.1 Using a global address space

The symbolic processing using RISCs (SPUR) avoids synonyms by giving each process access to four disjoint 1G-byte segments that are mapped onto a 256G-byte global virtual address space [28]. No synonyms are allowed to exist in the global address space. Instead, memory is shared between processes at the granularity of segments. This avoids synonyms but restricts sharing to a course granularity and disallows synonyms within a single virtual address space. As modern operating systems rely on such features, this solution is not very portable.

3.7.2 Synonym Alignment

Synonyms can be prevented in software by ensuring that synonyms map to the same set in the virtual caches (*alignment*). This forces the operating system to deal with synonyms instead. As modern operating systems have poor support for synonym alignment, this solution is not very portable. To prevent synonyms from coexisting within the same cache set, the cache must be either VIPT or direct-mapped.

3.7.3 Read-only synonyms

Another solution is to permit shared data to be read-only. Writes are permitted by flushing or invalidating any read-copies. This does not restrict sharing but has a negative impact on performance.

3.7.4 Flush virtual caches on context switch

Synonyms can be avoided by flushing the virtual caches on context switch. If the virtual caches are large, this can increase the miss frequency considerably. Write-back caches may need additional logic to spread the write-backs that occur on context switch through time (see Section 3.5.1).

3.7.5 Searching the superset

A brute force approach that works for physical tags is to perform tag comparison on all blocks in the superset. However, the required hardware area is proportional to the size of the superset and additional delay is introduced in tag comparison. Some systems have still used this approach, such as the IBM 3090 system [29] and the AMD Opteron 1000, 2000, and 8000 Series [30]. It is important that synonyms cannot coexist in the same set, therefore the cache must either be /VIPT or direct-mapped.

3.7.6 Backpointers in L2

A physical-address L2 cache can be used to store *backpointers* to a virtual L1 cache [21]. There is a backpointer for every block in the L1 cache, each pointing to the set in the cache where the block is valid. Synonyms are detected whenever an access misses in the L1 cache and the L2 cache contains a valid backpointer for that reference (*soft miss*). The block is then moved to the new cache set in L1 and retagged. To ensure that the L2 cache always contains backpointers for all synonyms, the L2 cache needs to be inclusive with respect to the L1 cache. Using backpointers for the L2 cache is not sufficient, thus the L2 cache cannot be virtual using this technique.

3.7.7 Dual-directory cache

Synonyms can be detected dynamically by extending the cache with a dual directory [17]. The dual directory is used to access cache blocks with physical addresses. On a miss, the virtual address is translated by the TLB and the cache is accessed through the dual directory to detect whether the accessed block was brought into the cache through a synonym. If present, the block is moved to the cache block selected through the virtual and the dual directory is updated; otherwise it is fetched from memory.

The dual directory can be used for reverse translation as part of cache coherency protocols such as snooping. Also, it can enable retranslation of virtual addresses on page-mapping changes. The main disadvantage is the additional hardware required to implement the dual directory. As when searching the superset, it is important that synonyms cannot coexist within the same set, therefore the cache must be either VIPT or direct-mapped. Also note that a hit through the dual directory takes more tag cycles that a direct hit because the cache is effectively accessed twice.

3.7.8 U-cache

The *U-cache* improves over the dual-directory cache by only providing reverse translations for unaligned synonyms [31]. This allows the U-cache to be smaller. The U-cache is used in a similar fashion to the dual-directory. Cache hits are processed as normal. On a cache miss, the U-cache is accessed to detect whether the accessed block resides in the cache in a different location. On a U-cache hit, the block is moved to the cache block selected through the virtual address and the U-cache pointer is updated. On a U-cache miss, either there are no unaligned synonyms in the cache, or there is an aligned synonym. The former case implies that the access is unaligned, otherwise the access would have resulted in a cache hit. To avoid inconsistencies, the cache block that may contain the aligned synonym must be checked. If an aligned synonym is present, it is used to service the cache miss; otherwise the missed block is fetched from main memory and the U-cache is updated with a pointer to it.

Similar to the dual-directory, the U-cache can be adapted to work with cache coherency protocols. The U-cache can benefit from page alignment performed by the operating system, but this is not necessary for correctness. When new entries are allocated in the U-cache, old entries are evicted.

3.7.9 Synonym Lookaside Buffer

A synonym lookaside buffer (TLB) can be used to avoid synonyms even with a virtual L2 cache [32][33][34]. An extended virtual address space is formed by extending the virtual addresses of the private address spaces with ASIDs. Every page is associated with a single unique extended virtual address called the *primary virtual address*; synonyms of the primary address are called *secondary virtual addresses*. Only primary addresses are ever used to access the memory hierarchy

Synonym Solution	Requirements	Performance Considerations
Flush cache on context switch	None significant	Flush virtual caches on context switch
Search the superset	Additional tag comparison hardware, cache must be VIPT or direct-mapped	Slower tag comparison
Backpointers in L2	Backpointers stored in L2 cache, L2 cache is physical	Slower access to synonyms
Dual directory cache	Dual directory, the L1 cache must be either VIPT or direct-mapped	Slower access to synonyms
U-cache with synonym alignment	U-cache (smaller than the dual directory), the L1 cache must be either VIPT or direct-mapped	Slower access to synonyms
Synonym lookaside buffer	Virtual L1 cache, small synonym lookaside buffer, software managed TLB and SLB	Flush SLB when the content of a virtual page is changed

Table 3.6: Comparison of techniques to solve the synonym problem.

(possibly excluding the L1 cache); a TLB is used to dynamically translate synonyms into their corresponding primary address on access. The TLB needs to be managed in software. When the operating systems detect synonyms for a page, one of the synonyms is selected as a primary virtual address. The mechanism for loading entries into the TLB works as follows. When the TLB cannot provide a translation for a synonym, the access is allowed to proceed down the memory hierarchy. The operating system ensures that such accesses will generate a TLB miss by not loading mappings for synonyms into the TLB. The TLB miss traps the processor, the missing primary virtual address can be loaded into the SLB, and memory access is retried with the primary address.

The TLB has better coverage than a TLB and can be kept small and fast (16 elements can be sufficient), thus it is possible that it can be accessed before the L1 cache. If parallel access is necessary, backpointers can be stored in the L2 cache to avoid aliases. The main drawback of using an TLB is that it requires specific support by the operating system.

Page-mapping operations affect the TLB and the L1 cache as usual. The SLB is only affected by rare mapping operations that change the content of a virtual page; then the TLB must be flushed for demapped secondary virtual addresses.

Chapter 4

Virtual Memory in SHMAC

This section discusses virtual memory systems with regards to the SHMAC architecture.

4.1 TLB Placement

The TLB can be placed either before or after the L1 caches; in the former case, the cache is either PIPT or VIPT, and in the latter case the cache must be VIVT.

4.1.1 TLB before a PIPT L1 cache

A potential disadvantage of locating the TLB in front of a PIPT L1 cache is that the TLB is placed on the critical path of cache access, either fully by accessing it in before accessing the cache or partly by accessing it in parallel with cache indexing. However, the SHMAC processor core is implemented using slow BRAM and runs at a relatively slow clock frequency when compared to modern processor cores. Since the cache is accessed in two stages, the latency of each stage is quite small. Consequently the core frequency is limited by the BRAM frequency, not the cache latency [15]. It is conceivable that the additional delay introduced by the TLB will not affect limit the core frequency regardless of whether it is accessed in serial or in parallel with the cache. If parallel access is used, the L1 caches currently are sufficiently small that the size of their index is not restricted [15]. Physical caches also have benefits when compared to virtual caches: aliases do not occur, translation information need not be buffered in the cache, and reverse translation hardware is not needed to support coherency protocols.

4.1.2 TLB after a Virtual L1 cache

The main disadvantages with virtual caches are that aliases must be handled and that reverse translation might be needed to support some coherency protocols. A potential benefit of locating the TLB after the cache is that the TLB is removed from the critical path and that a large TLB can be used to reduce the TLB miss frequency. As the critical path issue may be irrelevant, it is possible that a sufficiently large TLB can also be located before the cache to obtain an acceptable miss frequency.

4.1.3 TLB at the Memory Tile

Locating the TLB before the L2 cache at the memory tile poses some interesting issues due to the fact that the L2 cache can be distributed. To avoid defeating the benefits of using a distributed L2 cache the TLB must also be distributed. An obvious approach is to let each cache have its own TLB that contains translation information for the data present in that cache. One must consider whether different TLBs should be allowed to contain the same mappings. This necessitates the use of a distributed algorithm to flush invalid entries on page-mapping changes. It is not possible to use straight-forward TLB shootdown because the TLBs are no longer associated with individual cores. One might imagine laying a bus over the grid to allow TLB coherency transactions, either initiated by the cores on a page-mapping change or initiated by the caches when a new page frame is loaded to maintain exclusiveness, but the scalability of bus-based approaches in tile-based architectures is questionable. The grid can be used for communication, but this may in an increase in data traffic. The implementation would be simplified if mappings could only reside in a single TLB at once. This may be achieved by ensuring that only a page frame can only be present in a single cache. The operating system must track the the location of different page frames in a data structure in order to know which TLB to flush at a page-mapping change. A handler is trapped on an L2 miss to update to the data structure. This approach limits the granularity with which data can be located in different caches to the size of a page.

4.2 Process Isolation

Four different techniques for process isolation were introduced in Section 3.5, two for private page tables and two for global page tables.

An important factor that sets private and global page tables apart is that contemporary operating systems have poor support for global page tables. Private page tables have been the dominating trend. Building hardware that relies on using a global page table will inevitably increase the efforts necessary to port operating systems onto the SHMAC.

When private page tables are used, either the TLB and any virtual caches must be flushed on context switch or the entries in the TLB and the virtual caches must be extended with an ASID field. Because the caches currently used in SHMAC are relatively small, flushing may be an acceptable alternative. However, it is also possible that future versions of the SHMAC will need larger L1 caches in order to accommodate the memory needs of a modern operating system running multiple processes. ASIDs can be used to avoid flushing at context switch, but this will increase the area of the TLB and virtual caches; at the moment, area is a limited FPGA resource [15].

4.3 Solving the Synonym Problem

The synonyms problem must be solved when virtual caches are used. The TLB can be located before a VIPT or direct-mapped L1 cache, after a virtual L1 cache, or on the memory tile. When the TLB is located on the processor tile, a simple solution is to flush the L1 caches on context switch, possibly with a negative impact on performance (see Section 4.2).

Some solutions require that a VIPT or direct-mapped L1 cache is used. Searching the superset on access may increase the area used for the tag comparison hardware and consume more FPGA resources. Storing backpointers in the L2 cache may significantly increase memory traffic in the grid as it would then be necessary to access the L2 cache once for every synonym access. A dual directory uses a considerable amount of hardware area by implementing a reverse translation table for all cache entries, although it provides some support for cache coherence protocols. The U-cache is similar to the dual directory but only provides reverse translation for a configurable number of cache entries that are synonyms. Thus it offers a trade-off between area and performance but no support for cache coherence protocols.

An TLB can be used to support VIVT caches but since software keeps track of synonyms a TLB may significantly increase the effort needed to port operating systems onto the architecture.

When the TLB is located on the memory tile, storing backpointers in L2 may be a more promising alternative as it does not increase memory traffic.

Chapter 5

Conclusion

There are several promising virtual memory system designs for the SHMAC.

The simplest design is to place a TLB in front of a PIPT L1 cache and access it either in sequence or in parallel. This allows for a simple implementation with no synonyms and no impact on the coherence protocol. The potential impact on core frequency should be determined and an algorithm must be devised that will work on grid-based architectures, for instance by adapting the TLB shootdown algorithm.

An alternative design is to place the TLB after a VIVT L1 cache. The main motivation for this technique in SHMAC is that since the TLB is no longer on the critical path it can be large and slow to support a good hit rate. Also, using virtual address bits to index the L1 cache allows compilers to optimize the placement of data in the cache. A significant disadvantage is that synonyms in the virtual cache must be handled and that cache coherence protocols might need reverse translation to invalidate cache data. The potential performance benefits of this scheme would need to be estimated and compared against the simpler scheme of using a PIPT L1 cache. If the differences favor the virtual cache approach, a SHMAC-specific solution to the synonym problem and the cache coherency problem could be devised.

A more novel design to virtual memory is to place the TLB at the memory tile. Since virtual addresses are used on the bus this algorithm has no impact on the cache coherency protocol. It is necessary to investigate whether the TLB can meet the bandwidth requirements set by the processing cores. Also, since the L2 cache can be distributed over multiple tiles, some scheme must be devised to implement a distributed TLB.

A very simple technique that can be used both to implement process isolation and solve the synonym problem is to flush the L1 caches on context switch. The potentially negative performance impact associated with flushing the caches on the SHMAC should be investigated to determine whether this is a viable technique when implementing virtual memory in the SHMAC.

Bibliography

[1] Gordon Moore. Cramming more components onto integrated circuits, electronics,(38) 8, 1965.

[2] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[3] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[4] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[5] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[6] The EECS Group at NTNU. The single-isa heterogeneous many-core computer (shmac). http://www.ntnu.edu/ime/eecs/shmac, 2016.

[7] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[8] The RISC-V Foundation. Risc-v: The free and open risc instruction set architecture, 2016.

[9] Bruce Jacob and Trevor Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, 1998.

[10] Bruce Jacob and Trevor Mudge. Virtual memory: Issues of implementation. *Computer*, 31(6):33–43, 1998.

[11] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface.* Newnes, 2013.

[12] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[13] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, et al. Tile64-processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598. IEEE, 2008.

[14] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. The risc-v instruction set manual. *volume I: User-level ISA, version 2.0, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, 2014.

[15] Magnus Jahre and Asbjørn Djupdal. private communication.

[16] Michel Cekleov and Michel Dubois. Virtual-address caches. part 1: problems and solutions in uniprocessors. *IEEE Micro*, 17(5):64–71, 1997.

[17] James R Goodman. Coherency for multiprocessor virtual address caches. *ACM SIGARCH Computer Architecture News*, 15(5):72–81, 1987.

[18] Eric J Koldinger, Jeffrey S Chase, and Susan J Eggers. Architecture support for single address space operating systems. *ACM SIGPLAN Notices*, 27(9), 1992.

[19] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGARCH Computer Architecture News*, 19(2):63–74, 1991.

[20] David A Wood, Susan J Eggers, Garth Gibson, Mark D Hill, and Joan M Pendleton. An in-cache address translation mechanism. *ACM SIGARCH Computer Architecture News*, 14(2):358–365, 1986.

[21] Wen-Hann Wang, J-L Baer, and Henry M Levy. Organization and performance of a two-level virtual-real cache hierarchy. *ACM SIGARCH Computer Architecture News*, 17(3):140–148, 1989.

[22] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[23] Ruby B Lee. Precision architecture. *Computer*, 22(1):78–91, 1989.

[24] Michel Cekleov and Michel Dubois. Virtual-address caches. 2. multiprocessor issues. *IEEE Micro*, 17(6):69–74, 1997.

[25] David L Black, Richard F Rashid, David B Golub, and Charles R Hill. Translation lookaside buffer consistency: a software approach. In *ACM SIGARCH Computer Architecture News*, volume 17, pages 113–122. ACM, 1989.

[26] Patricia J. Teller. Translation-lookaside buffer consistency. *Computer*, 23(6):26–36, 1990.

[27] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 340–349. IEEE, 2011.

[28] Mark Hill, Susan Eggers, Jim Larus, George Taylor, Glenn Adams, BK Bose, Garth Gibson, Paul Hansen, Jon Keller, Shing Kong, et al. Design decisions in spur. *Computer*, 19(11):8–22, 1986.

[29] Stuart G. Tucker. The ibm 3090 system: An overview. *IBM Systems Journal*, 25(1):4–19, 1986.

[30] John Levesque, Jeff Larkin, Martyn Foster, Joe Glenski, Garry Geissler, Stephen Whalen, Brian Waldecker, Jonathan Carter, David Skinner, Helen He, et al. Understanding and mitigating multicore performance issues on the amd opteron architecture. Technical report, Berkeley Lab Scientific Publications, 2007.

[31] Jesung Kim, Sang Lyul Min, Sanghoon Jeon, Byoungchu Ahn, Deog-Kyoon Jeong, and Chong Sang Kim. U-cache: a cost-effective solution to synonym problem. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 243–252. IEEE, 1995.

[32] Xiaogang Qiu and Michel Dubois. Towards virtually-addressed memory hierarchies. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 51–62. IEEE, 2001.

[33] Xiaogang Qiu and Michel Dubois. Moving address translation closer to memory in distributed shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):612–623, 2005.

[34] Xiaogang Qiu and Michel Dubois. The synonym lookaside buffer: A solution to the synonym problem in virtual caches. *IEEE Transactions on Computers*, 57(12):1585–1599, 2008.