Memory Hierarchy Extensions to the SimpleScalar Tool Set

Doug Burger[†] Alain Kägi[‡] M.S.Hrishikesh[*]

Computer Architecture and Technology Laboratory
[†]Department of Computer Sciences
[*]Department of Electrical and Computer Engineering
[‡]Microprocessor Research Lab, Intel Corporation
Tech Report TR99-25
The University of Texas at Austin
cart@cs.utexas.edu — www.cs.utexas.edu/users/cart

Abstract

In this report we describe memory hierarchy extensions to the Simplescalar tool set. The extensions allow for the modeling of an arbitrary hierarchy of caches and associated buses. The caches are non-blocking caches with a finite number of miss status holding registers and support both virtual and physical addressing. We also model detailed simulation of address translation, the hardware translation look-aside buffer (TLB) fills and page table walks. In this report we describe the cache and memory organization that we model and present the implementation in detail. We also describe how these extensions can be configured and provide a sample configuration.

1 Overview

The Simplescalar tool set [1] simulates a superscalar, out-of-order processor in detail, along with a simple model of the memory hierarchy (versions 2.0 and 3.0), including caches that allow an infinite number of outstanding misses and no modeling of bus contention and latency. Also, the memory hierarchy is not easily configurable to model hierarchies more complex than 2-levels of inclusive caches. We added modifications to the Simplescalar tool set to provide a more realistic model of the memory hierarchy. In this report we describe our modifications and show how the new model can be configured.

These memory extensions model a non-blocking cache that allows only a finite number of outstanding misses and allows for flexible configuration of the memory hierarchy. An arbitrarily deep cache hierarchy can be modeled by simply changing the configuration parameters. In addition, we model translation look-aside buffers (TLBs), simulate page-table walks and the caching of page table entries. We also model the buses between multiple levels in the memory hierarchy. Currently our memory extensions support only the sim-outorder simulator executing PISA binaries. Other ISAs will be supported in the next release of the memory extensions.

The rest of this report is organized as follows. Section 2 describes how different components of the memory hierarchy are configured and shows a sample configuration. Section 3 describes the implementation details of the new memory extensions.

2 Configuring and Using the Memory Hierarchy

2.1 Configuring Caches and TLBs

```
-cache:dcache <name>
-cache:icache <name>
-cache:define <name>:<nsets>:<bsize>:<subblock>:<assoc>:<repl>:<hitlatency>:
              <translation>:<prefetch>:<# resources>:<resource code>:[resource names]*
```

The memory configuration requires the names of the level-1 caches to be explicitly specified. The syntax to define the names of the level-1 instruction and data cache is shown above. A cache configuration line begins with the flag "-cache:define" and is followed by the parameters of the cache separated by colons. The first parameter defines the name associated with the cache. This name will be used to identify the cache in the configuration file. The next four parameters are the number of sets, the size of a cache line, the number of sub-blocks, and the associativity of the cache. The cache capacity is the product of nsets, bsize and assoc.

The other cache parameters are the replacement policy, cache hit latency, the cache translation and a prefetch flag. The cache replacement policy determines which line in the cache will be evicted to make room for a new cache line. There are three cache replacement policies that are supported: LRU (l), Random (r) and FIFO (f). The replacement parameter should be set to "l", "r" or "f" to select the appropriate policy.

The cache translation parameter is used to simulate one of the following configurations: virtually indexed virtually tagged (VIVT) virtually-indexed physically-tagged (VIPT) and physically-indexed physically-tagged (PIPT). Physically-indexed virtually-tagged caches are not supported. All cache configurations other than VIVT require the virtual address (from the processor) to be translated to a physical address by performing a TLB lookup. Only the highest level of physically-tagged or physically-indexed cache triggers an address translation. For VIPT configurations the cache can be indexed using the virtual address and only the tag comparisons have to wait until the physical address is available. Therefore for this configuration the cache access and the TLB access occur in parallel. However, for PIPT configurations the TLB is first accessed to obtain a physical address which in turn is used to access the cache.

The pre fetch parameter is a flag that determines if the cache pre fetches data. If this flag is set to 1 then on a cache hit the next contiguous cache line is pre fetched [2]. The resources parameter sets the number of buses that connect the cache to the hierarchy below. It is intended to be used for specifying different ways of mapping requests to multiple resources. Currently, each cache can be connected to the memory/cache hierarchy below via a single bus. Therefore, this parameter should be set to 1 if the cache is connected to a bus. The next parameter (resource code) should be set to 0, and is used for selecting policies for choosing one of multiple buses, if the cache connects to more than one. The last parameter defines the names of the buses that the cache is connected to.

Each cache has a fixed number of *miss status holding registers* (MSHRs) associated with it. The number of MSHRs and the number of targets per register are specified as shown below. These configuration parameters specify the number of MSHRs in every cache for the hierarchy.

```
-cache:mshrs <number of mshrs>
-cache:mshr_targets <number of targets>
```

Translation look-aside buffers (TLBs) are configured as virtually-indexed virtually-tagged caches. The other parameters for TLBs are identical to the cache parameters. The names of the instruction and data TLBs must be explicitly specified as shown below.

```
-tlb:dtlb  <name>
-tlb:itlb  <name>
```

2.2 Configuring Buses and Memory

```
-bus:define <name>:<width>:<cycle_differential>:<arbitration penalty>:<inf bandwidth>:
    <# resources>:<resource code>:[resource names]*
```

The syntax to configure a bus is shown above. The first parameter defines the name associated with the bus. This name will be used to refer to the bus in the configuration file. The other parameters are the width of the bus (in bytes), the number of processor cycles that equals one bus cycle, the bus arbitration penalty (in bus cycles) and a flag to simulate infinite bandwidth with zero contention delays. The resources and resource code parameters are similar to the parameters in the cache definition. The resource name parameter should be set to the name(s) of the memory structure(s) or cache(s) to which the bus connects.

```
-mem:define <name>:<access time>:<access code>:<cycle ratio>
```

The syntax to configure the parameters for memory is shown above. The first parameter defines a label that will be used within the configuration file to refer to a memory unit. The second parameter is the access latency in cycles (at bus frequency). The third parameter defines the type of memory (SDRAM, RAMBUS etc). Currently only SDRAM is supported and this parameter should be set to 0; future extensions will support other types of memory. The last parameter defines the ratio between the memory bank frequency and the processor frequency.

2.3 Sample Configuration

Below is a sample configuration. The configuration defines a memory hierarchy consisting of split level-1 caches backed by a unified level-2 cache and a level-3 cache. The level-1 data cache (DL1) is 8KB large, 2-way associative and virtually indexed physically tagged. It has a 3-cycle access latency and uses the FIFO replacement policy. The level-1 instruction cache (IL1) is 64KB large, 2 way associative, virtually-indexed virtually-tagged and uses LRU replacement policy. The IL1 cache has a one cycle hit latency. Both the level-1 caches are connected to the level-2 cache via a common bus (L2bus) that is 16 bytes wide. The level-2 cache is 2MB with a 6-cycle access latency while the level-3 cache is 4MB large with a 14-cycle access latency. Both the L2 and L3 caches are direct mapped.

```
# defines name of first-level data cache
-cache:dcache DL1

# defines name of first-level instruction cache
-cache:icache IL1

# defines the names of the data and instruction TLBs
-tlb:dtlb DTLB
-tlb:itlb ITLB


# cache configurations
-cache:define          DL1:64:64:0:2:f:3:vipt:0:1:0:L2bus
-cache:define          IL1:512:64:0:2:l:1:vivt:1:1:0:L2bus
-cache:define           L2:32768:64:0:1:l:6:pipt:0:1:0:L3bus
-cache:define           L3:65536:64:0:1:l:14:pipt:0:1:0:Membus


# number of regular mshrs for each cache
-cache:mshrs 8

# number of prefetch mshrs for each cache
-cache:prefetch_mshrs 4

# number of targets for each MSHR entry
-cache:mshr_targets 8

# define tlbs
-tlb:define DTLB:1:32:0:128:l:1:vivt:0:1:0:L2bus
-tlb:define ITLB:1:32:0:128:l:1:vivt:0:1:0:L2bus

# bus configuration
-bus:define L2bus:16:1:1:0:1:0:L2
-bus:define L3bus:16:1:1:0:1:0:L3
-bus:define Membus:8:4:1:0:1:0:SDRAM
```

```
# memory bank configuration
-mem:define SDRAM:150:0:1

# Additional options

# flush caches on system calls
-cache:flush false
```

3 Implementation

This section describes the implementation of the memory hierarchy extensions. We first present details of the data structures used to model the memory hierarchy and then describe the relevant functions that simulate memory access.

3.1 Data Structures

3.1.1 Cache Access Packet Structure

The *cache_access_packet* is a structure used to pack the parameters that define each cache access. The important parameters in cache_access_packet are shown below. This structure stores a pointer to the cache structure being accessed, the address to access, a flag indicating if the address is virtual, the number of bytes required and the type of access (Read/Write). In addition, cache_acess_packet also contains pointers to two functions — the release function (*release_fn*) and the valid function (*valid_fn*). These functions are "callback functions" and are used if the access results in a cache miss or cannot be served immediately. On a cache miss subsequent structures (cache/memory) in the hierarchy are accessed until the miss is resolved. At that point the valid function is used to check if the instruction that generated this cache access is still valid (e.g. it has not been squashed due to a pipeline flush). If the instruction is still valid, the release function is called. For example, the release function could be used to inform a load/store operation that the access it initiated has completed.

```
typedef struct _cache_access_packet
{
  struct cache *cp; /* cache to access */
  unsigned int cmd; /* access type, Read or Write */
  md_addr_t addr; /* address of access */
  enum trans_cmd vorp; /* is the address virtual or physical*/
  int nbytes; /* number of bytes to access */
  RELEASE_FN_TYPE release_fn; /* Function to call upon cache release */
  VALID_FN_TYPE valid_fn; /* Function to check validity of return */
} cache_access_packet;
```

3.1.2 MSHR Structure

The cache data structure from Simplescalar 3.0 was modified to support the new memory extensions. The most significant change is the addition of a structure to support MSHR registers. The important parameters in the MSHR structure are shown below. The MSHR registers hold state information for accesses that missed in the cache. Each MSHR entry stores the following data — address of the missed access, the number of bytes requested and the type of the access (read/write). A specified number of cache misses to the same cache line, can be coalesced into one MSHR register by allocating multiple targets. The *ntargets* variable holds the number of misses that are coalesced into each MSHR entry.

```
struct mshregisters
{
  md_addr_t addr; /* address sent to next cache level */
  unsigned int cmd; /* Read if all targets are reads */
  unsigned int size; /* Number of bytes requested */
  int ntargets; /* number of allocated targets */
  struct target_table
  {
    tick_t time; /* time of request */
    struct _cache_access_packet *pkt; /* packet representing this cache access */
  } target_table[MAX_TARGETS];         /* target descriptors */
} mshregisters[MAX_MSHRS];
```

3.2 Cache Access Functions

In this section we describe in detail the functions that are used to simulate accesses to the memory hierarchy.

3.2.1 Simulating Cache Access

A cache access is initiated by first creating a cache_access_packet and passing it as an argument to the cache_timing_access function to access the level-1 cache. For example, the ruu_issue() function initiates a cache access for loads by creating a cache_access_packet to access the level-1 data cache. The cp variable in the cache_access_packet is set to point to the level-1 data cache structure, the valid_fn variable is set to the valid_rs() function, and the release_fn variable is set to eventq_queue_event(). The valid_rs() function is used to check the appropriate RUU entry to ensure that the instruction is still valid (i.e. it has not been flushed from the pipeline). The eventq_queue_event() function is used to place an instruction onto a queue of completed instructions.

Figure 1 shows a flow chart of a call to the cache_timing_access() function. The function first checks to see if the address is a virtual address. If the address is virtual and the cache being accessed is not *virtually indexed and virtually tagged* (VIVT) then the cache packet is passed to the cache_translate_address() function to translate the virtual address to a physical address. The working of cache_translate_address() is described later in this section. Once the address is translated, the *tag* bits from the address are extracted and compared with the tags of the the appropriate cache set. If the comparison indicates a cache hit, the function returns the cache hit latency. On a cache miss, the MSHR registers are searched to find if a previous cache miss has occurred for the same cache line. If an MSHR match is found and an MSHR target entry is available the cache packet is added to the MSHR entry's list of targets. The function then returns a CACHE_MISS. If the matching MSHR entry has no free target entries available, the function

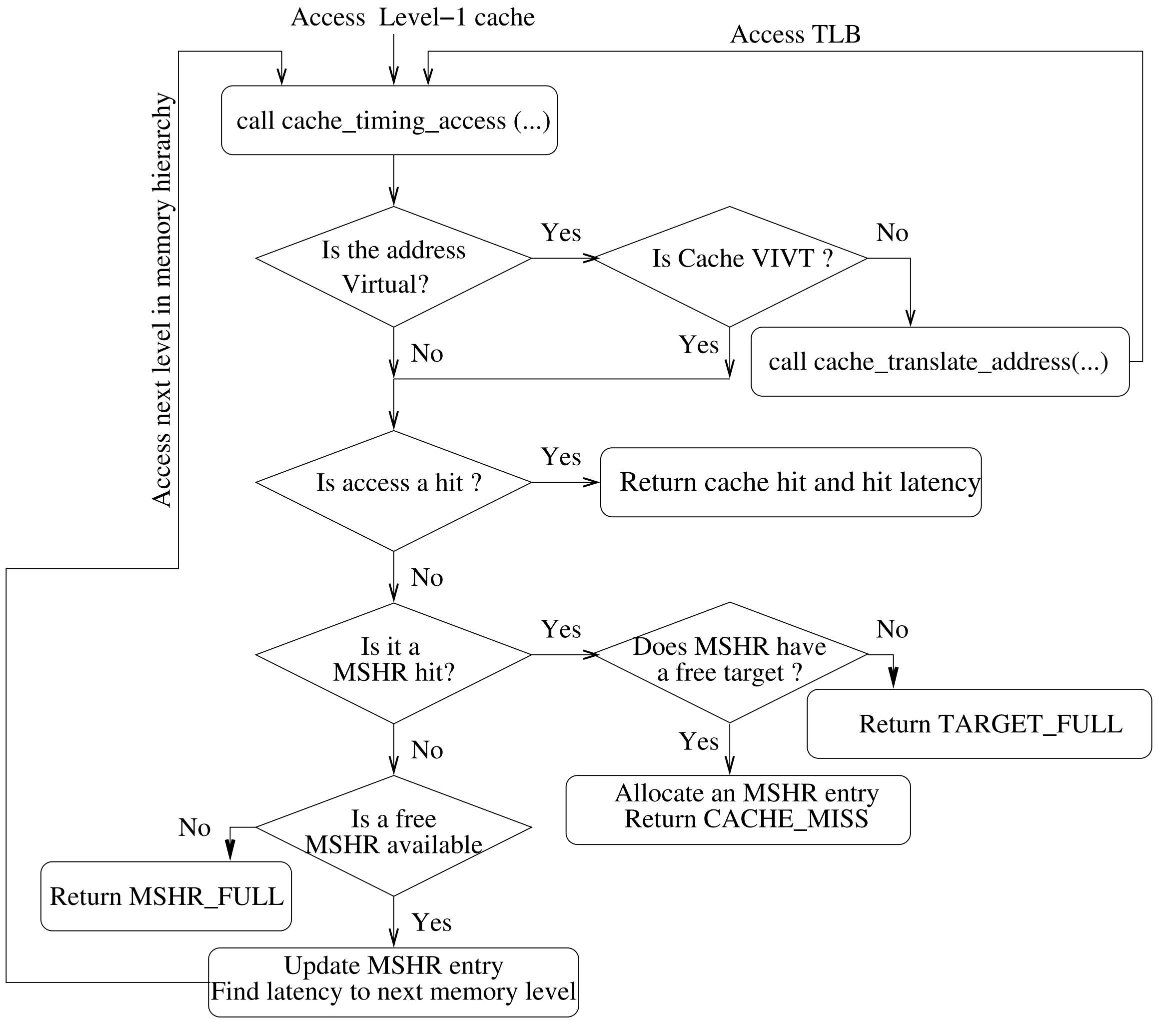

Figure 1: A flowchart showing the operations that are performed by the cache_timing_access function.

returns a value indicating that the MSHR targets are full and the access should be re-tried at a later time.

If there is no outstanding miss to the same cache line (an MSHR miss), a new MSHR entry is allocated and the cache packet is stored as a target in that entry. A new cache_access_packet is then created to access the next level of memory. The valid_fn and release_fn in the new cache_access_packet are set to valid_mshr() and schedule_response_handler() respectively. The bus to the next level of cache/memory is checked to determine the delays due to latency and contention. An event is then scheduled on the global event queue to access the next level of cache using the cache_timing_access function.

A missed access travels down the memory hierarchy via successive calls to cache_timing_access() and is eventually resolved at some level in the hierarchy (i.e. hits in cache or memory). Once the access is resolved the valid_fn from the cache cache_access_packet is invoked to check if the instruction that initiated this memory operation is still valid. If the instruction is still valid the release_fn (schedule_response_handler()) in the cache_access_packet is called. The schedule_response_handler() function calls the response_handler() function and passes it a pointer to the cache_access_packet that was used to access the previous cache in the hierarchy (i.e. the last level in the cache hierarchy that suffered a miss). The response_handler() function selects a line from the cache to evict based on the cache's replacement policy. If the selected line is valid and dirty then it is written to the lower level in the hierarchy and tags of the corresponding cache-line is set to the new value. The MSHRs are then searched to find the entry corresponding to the returning cache miss and the release functions of all the targets in the MSHR are called.

3.2.2 Address Translation

Virtual addresses are translated to physical addresses by accessing the TLB. The cache_translate_address() and the cache_timing_access() functions are used to simulate TLB accesses. As shown in Figure 1, when a non-VIVT cache is accessed with a virtual address then the address is passed to the cache_translate_address function for address translation. This function creates a cache_access_packet and accesses the TLB by invoking the cache_timing_access function. TLBs are organized like VIVT caches and accesses to the TLB happen as described in 3.2.1. On a TLB miss the translation is found by accessing the lower levels of the hierarchy.

3.2.3 Implementation Notes

This subsection lists the new files that have been added to support the memory hierarchy extensions. The cache.c file from the original Simplescalar (Version 3.0a) has been split into three files. Cache_timing.c contains the code to support the memory hierarchy extensions, cache_func.c contains code to support functional execution (sim-fast) and is largely unchanged from Simplescalar 3.0a, and cache_common.c contains code that are used by both sim-outorder and other simulators in the suite (e.g. code for cache eviction policies).

```
bus.c, bus.h                            : Code to model buses
cache_common.c, cache_func.c,
cache_timing.c cache.h                  : Code to model cache access
mshr.h                                  : Code to model MSHRs
tlb.c, tlb.h                            : Code to model TLB access
```

Note that the functions in cache_func.c are used by the sim-fast and sim-cache simulators. However, only sim-outorder is supported in this release. Other simulators will be supported in future releases.

4 Summary

In this report we describe memory hierarchy extensions to the Simplescalar tool set (version 3.0). These extensions enable modeling cache hierarchies of arbitrary depths and they also model the buses that connect the different levels. The caches can be configured to use either virtual or physical addresses. We model virtual to physical address translation by simulating a hardware TLB. Section 2 provides a detailed explanation of how the different memory components can be configured.

The memory extensions support only the sim-outorder simulator executing PISA binaries. Also, in the current version it is not possible to specify split TLBs for caches at level 2 and lower. All the caches that are at level 2 or lower use the DTLB (if specified) to perform address translation. Future versions will add additional support for other ISAs and to provide greater flexibility in configuring TLBs.

References

[1] D. Burger and T. M. Austin, "The simplescalar tool set version 2.0," Tech. Rep. 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[2] J. D. Gindele, "Buffer block prefetching method," *IBM Tech. Disclosure Bull.*, vol. 20, pp. 696–697, July 1977.