BusyMap, an Efficient Data Structure to Observe Interconnect Contention in SystemC TLM-2.0

Emad M. Arasteh
Chapman University
Orange, CA, USA
arasteh@chapman.edu

Vivek Govindasamy, and Rainer Dömer
Center for Embedded and Cyber-physical Systems
University of California, Irvine, USA
{*vbgovind,doemer@uci.edu*}

***Abstract*—For designing embedded computer architectures that meet desired performance constraints at low cost, fast and accurate simulation models are needed early in the design flow. To identify and avoid bottlenecks early at the system level, observing the contention of shared resources is critical. In this paper, we propose and evaluate a novel data structure called BusyMap that accurately reflects contention at system busses or similar interconnect components. BusyMap is an efficient data structure that allows the system designer to accurately model and easily observe contention in IEEE loosely-timed TLM-2.0. In contrast to prior state-of-the-art, our model fully supports temporal decoupling and multiple levels of interconnect. Our experiments demonstrate the effectiveness of BusyMap with results showing high accuracy at high-speed SystemC simulation.**

***Index Terms*—transaction-level modeling, temporal decoupling, systemc, memory contention, multi-core processor**

I. Introduction

The emergence of new data-intensive applications, such as artificial intelligence (AI), machine learning (ML), robotics, virtual reality, the Internet of Things (IoT), and personalized medicine, requires a significant increase in embedded computer systems' computation and storage capacity. Additionally, the energy efficiency requirements of embedded computer systems create extra design challenges to achieve on-par performance improvements. To overcome the challenges of designing today's complex system-on-chips (SoCs), electronic system level (ESL) methodology has been proposed for modeling systems at higher levels of abstraction early in the design flow [1], [2]. ESL ideas resulted in defining system-level description languages (SLDL), such as SpecC [3] and SystemC, the IEEE standard language for system modeling and simulation that can model hardware and software components and their detailed interactions [4].

With the ever-increasing demand for performance requirements of new embedded applications, one critical aspect of modeling and analyzing system models is finding resource contention early in the design flow. SystemC's loosely-timed coding style supports modeling hardware components, such as processors, interconnects, memories, timers, and interrupts, in a modern multiprocessor system on a chip (MPSoC). This coding style is suitable for software development with a virtual platform model of an MPSoC. This coding style also supports temporal decoupling, allowing individual SystemC processes to operate ahead of simulation time in a local "time warp" until they require synchronization with the rest of the system. This technique can lead to faster simulation for specific systems due to the enhanced data and code locality and the reduced scheduling overhead within the simulator.

A. Problem Definition

SystemC loosely-timed models provide a good trade-off between timing accuracy and simulation speed by modeling only two timing points for each transaction in a system: the start and end. However, resource contention is often detectable with increasing timing points for each transaction, which will result in significantly slower simulation. While there have been efforts to model contention in loosely-timed SystemC models, the approaches lack supporting temporal decoupling for faster simulation speed and multiple interconnect levels. A fast and accurate loosely-timed contention model allows the system designer to rapidly detect resource contentions and evaluate design candidates for lower-level implementations.

Our key contributions are as follows:

(1) Novel resource contention modeling in TLM-2.0 loosely-timed system models supporting temporal decoupling with high accuracy at high-speed simulation

(2) Support multiple-level hierarchical interconnects, including multi-level caches or multiple levels of buses

II. Related Work

System-level performance modeling and resource contention analysis have been widely studied under two broad categories: analytical and simulated-based approaches. Analytical approaches involve the mathematical modeling of systems and the derivation of their performance as a function of workload and input parameters [5], [6]. These analytical models depend on the program or the architecture modeled, and a new model must be developed for each new application, which requires a profound understanding of the application [7]. Furthermore, analytical modeling fails to consider the system's dynamic behavior [8].

Simulation-based techniques have the capacity to capture dynamic interactions within a given system. Two commonly utilized system-level description languages for the purpose of modeling, simulation, and validation of complex system-on-chip models are SpecC [3] and SystemC [9]. The SystemC C++ class library, an IEEE standard, facilitates system and transaction-level modeling through discrete event simulation (DES) [4]. However, prolonged simulator run times often hinder simulation techniques when operating at lower abstraction levels. Additionally, the manual construction and debugging

979-8-3503-4859-0/DATE24/© 2024 EDAA

of simulation models can result in significant design and development time.

To address the limitations of simulation-based methods, a hybrid approach that combines analytical and simulation methodologies has been proposed [7], [10], [11]. Although mixed methodologies have been employed to reduce simulator run times, the simulation's ability to cover corner cases remains challenging [11]. Aside from analytical and simulation-based modeling approaches, there are also statistical and stochastical techniques to capture the effect of resource contention [12], [13]. However, these approaches mainly operate at a significantly higher level of abstraction than transaction-level modeling, resulting in a trade-off between accuracy and simulation speedup.

Contention modeling in transaction-level abstraction has also been studied to break the speed-accuracy simulation trade-off. For instance, [14] proposes a TLM-2.0 loosely-timed contention-aware modeling (LT-CA) technique that offers high simulation speed with the accurate observation of memory contention in the system model. However, the approach does not support temporal decoupling and pipelined transactions. Moreover, the LT-CA described in [14] does not support contention modeling of hierarchical interconnects such as multiple levels of caches or buses in the system platform model.

III. Loosely-Timed Contention-Aware Modeling

The emphasis of transaction-level modeling (TLM) is mainly on the abstraction of data communication. In the TLM-2.0 loosely-timed (LT) modeling style, each transaction has two timing points: start and end. In this modeling style, transactions are completed in a single blocking transport method call, which simulates fast. However, observing resource contention often requires a detailed sequence of interactions in a transaction with more than two timing points. However, increasing the number of transaction timing points will significantly slow the simulation speed. Like conventional LT models, the loosely-timed contention-aware (LT-CA) modeling approach also uses a blocking transport call, which each transaction completes in a single method call. In contrast to conventional LT modeling, the LT-CA model style uses the timing annotation in the blocking transport interface to keep track of the contention status of resources in the transaction's path. By retaining the resource-busy status within a state variable located within the interconnect, it can effectively schedule transactions at the appropriate simulation time [14].

LT-CA approach benefits system designers and increases their productivity in multiple ways. First, it does not require implementing an elaborate call sequence with multiple *phases* of the transaction using non-blocking transport interface methods. Bug-free implementation of the transaction base protocol used in approximately-timed modeling without race-condition tends to be time-consuming and cumbersome. Second, LT-CA modeling does not require complex data structures such as Payload Event Queue (PEQ) to store pending transactions that can potentially slow the simulation speed. Finally, the LT-CA model can be effectively used to create an agile hardware/software co-design environment where the system designer can develop software by taking the effect of critical aspects of resource contention early in the design flow. Given the significant benefits of the LT-CA approach, it is not intended to replace approximately-timed or cycle-accurate simulations.

The LT-CA model can effectively model system platforms with multi-cores and shared interconnect and memory. As depicted in Fig. 1, a 16-core multi-core architecture connected to the shared bus interconnect and memory is modeled using TLM-2.0 sockets, generic payloads with memory addressing and transaction timings.

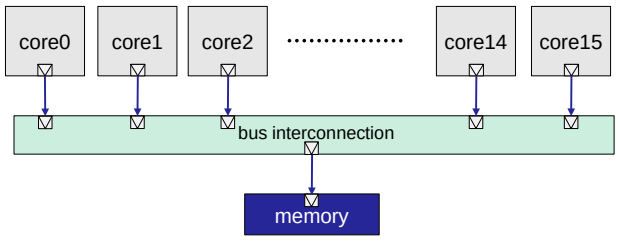


Fig. 1: SystemC TLM-2.0 platform model of simplified multi-core architecture with bus interconnect and shared memory (adapted from [14])

In this work, we will extend the approach by also supporting multi-level hierarchies and caches as illustrated in Fig. 2.

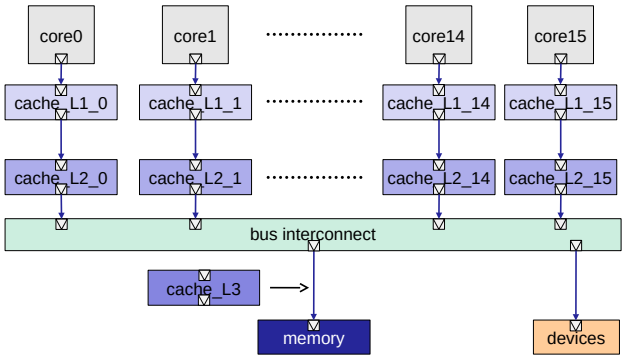


Fig. 2: SystemC TLM-2.0 virtual platform model of symmetric multiprocessing system (SMP) with multi-level caches and shared memory

A. Bus Contention using `busy_until` [14]

Before we describe our novel BusyMap approach for LT-CA modeling, we briefly review the existing work [14] as a reference for later comparison. The published approach [14] is very simple yet effective. It uses a single time stamp `busy_until`, which keeps track of the time a resource is busy. We replicate this approach in Algorithm 1 (with minor modifications[1]) for easier comparison. The bus module forwards incoming transactions to a memory or other targets after performing address translation. It observes the delay incurred

[1]The original component [14] supports only one outgoing target socket (essentially acts as a multiplexer), but we use it here as a bus that routes packets to different targets.

in the target (`memory_delay`) and uses that to maintain bus contention. Here, the bus is considered busy for its own `bus_delay` plus the `memory_delay`. This is recorded in the status variable `busy_until`. The remaining time left until the bus and memory become available again (`busy_delay`) is added to the cumulative contention of the bus during the simulation. Finally, the timing annotation of the transaction is extended with the `bus_delay` (latency of address translation and routing) and `busy_delay` (time to wait for bus access due to other transactions being processed).

Algorithm 1: Modeling bus contention using a time stamp `busy_until` (adapted from [14])

```
Module Bus_BusyUntil begin
    target_socket S_in[NUM_TARGETS];
    initiator_socket S_OUT[NUM_INITIATORS];
    time bus_delay;
    time contention := 0;
    time busy_until := 0;
    Procedure ForwardRequest (trans, delay) begin
        // perform address translation
        socket := decode_and_translate(trans.address);
        // forward the transaction
        d1 := delay;
        socket→b_transport(transaction, delay);
        d2 := delay;
        memory_delay := d2 - d1;
        // maintain bus contention
        busy_span := bus_delay + memory_delay;
        busy_until := max(busy_until, global_time);
        busy_delay := busy_until - global_time;
        busy_until += busy_span;
        contention += busy_delay;
        delay += bus_delay + busy_delay;
    end
end
```

While this approach supports a First-Come-First-Served (FCFS) arbitration policy[2], it cannot handle temporal decoupling because it relies on `global_time` being current at all times. Looking closer into [14], we note two assumptions that must hold:

1) Components after the bus (outgoing `b_transport` calls to the initiator sockets) must never call `wait` because such delay would not be observed.
2) Components before the bus (incoming `b_transport` calls from the target sockets) must always call `wait` to keep current with the `global_time`.

While the first assumption is desirable and necessary for speedy LT-CA simulation, the second assumption is too strict because it forbids temporal decoupling. With temporal decoupling, initiators use the interconnect *out-of-order* with different `delay` offsets from the `global_time`. We will address this short-coming of [14] in the next section.

IV. BusyMap

The BusyMap data structure is an ordered map of key-value pairs (k, v). Both keys and values are of type time (`sc_time` in SystemC). As illustrated in Fig. 3, the key k specifies the start time when the resource becomes busy. The corresponding value v specifies the duration of how long the resource is used. In the example, the resource is busy from the start time 0 until time 3, then available until time 5, when it becomes busy again for the duration of 2 time units, and so on. Note that the busy periods $[k, v)$ include the start time indicated by the key k and last up to time $k + v$ (exclusively).

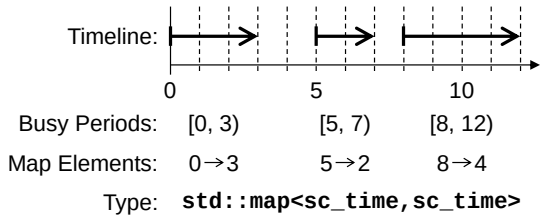


Fig. 3: BusyMap data structure, an ordered map of (start,duration) pairs

Since BusyMap is an ordered map, busy periods can be maintained efficiently during simulation. Whenever a resource is used, a corresponding reservation is entered into its BusyMap. Note that due to temporal decoupling, reservations can come in out-of-order. The complexity of finding or inserting a new busy period into the data structure is $O(\log n)$ where n is the number of existing map elements. This holds true even when we merge busy periods to keep the size of the data structure at a minimum. For example, if in Fig. 3 the resource is reserved at time 3 for 1 time unit, we extend the preceding map element $[0, 3)$ to $[0, 4)$ (`busy_map[0]=4;`) rather than inserting a new element $[3, 4)$. Also, if an available period is reserved entirely, for example, time $[7, 8)$, we merge the two adjacent periods $[5, 7)$ and $[8, 12)$ into a single one, i. e. $[5, 12)$.

When the simulator increases the system time, we adjust the BusyMap data structure accordingly to keep its size at minimum[3]. Specifically, we delete all elements with time periods before the current system time. Here, if the first map element (k, v) overlaps the current time t, we shorten it to $(t, v - (t - k))$.

A. Bus Contention using BusyMap

To observe and reflect contention in TLM-2.0 models with temporal decoupling, we place a BusyMap instance inside arbitrated interconnect components. Algorithm 2 lists the pseudo-code of a bus module with its essential member variables and methods. In contrast to Algorithm 1 with the `busy_until` member, the BusyMap algorithm contains the ordered `busy_map`.

The method ForwardRequest implements the bus functionality. It decodes and translates addresses before forwarding the `b_transport` call. For the latter, it considers any `delay` offset in addition to `global_time` due to temporal decoupling. This produces an accurate `memory_delay`, regardless of

[2]A Round-Robin approximation is also supported [14], but not relevant here.

[3]Unless we want to use older BusyMap elements for tracing purposes or other analysis, we remove expired elements as soon as possible to minimize the complexity of the data structure maintenance operations.

Algorithm 2: Modeling bus contention using BusyMap

```
Module Bus_BusyMap begin
    target_socket S_in[NUM_TARGETS];
    initiator_socket S_OUT[NUM_INITIATORS];
    time bus_delay;
    time contention := 0;
    ordered_map<time,time> busy_map;
    Procedure ForwardRequest (trans, delay) begin
        // perform address translation
        socket := decode_and_translate(trans.address);
        // forward the transaction
        d1 := global_time + delay;
        socket→b_transport(transaction, delay);
        d2 := global_time + delay;
        memory_delay := d2 - d1;
        // maintain bus contention
        busy_span := bus_delay + memory_delay;
        AdvanceTime();
        available_slot := FindFreeSlot(d1, busy_span);
        busy_delay := availabe_slot - d1;
        SetBusy(available_slot, busy_span);
        contention += busy_delay;
        delay += bus_delay + busy_delay;
    end
    Procedure AdvanceTime begin
        if busy_map.empty() then
            return;
        end
        keep := busy_map.upper_bound(global_time);
        if keep = busy_map.begin() then
            return;
        end
        cut := prev(keep);
        if cut→start = global_time then
            busy_map.erase(busy_map.begin(), cut);
            return;
        end
        if (cut→start + cut→duration) > global_time then
            cut_duration := cut→start + cut→duration -
              global_time;
            busy_map.erase(busy_map.begin(), keep);
            busy_map[global_time] := cut_duration;
        else
            if keep = busy_map.end() then
                busy_map.clear();
            else
                busy_map.erase(busy_map.begin(), keep);
            end
        end
    end
end
```

whether components after the bus call `wait` or not (condition 1 in Section III-A).

Bus contention is then maintained similar to Algorithm 1 but with three helper functions using the `busy_map`. First, AdvanceTime updates the `busy_map` to keep it in sync with any advances in `global_time`. As mentioned above, any map elements before `global_time` are erased or cut short to keep the data structure's size minimal.

Second, FindFreeSlot listed in Algorithm 3 finds the first available time where a period [*earliest*,*span*) can be reserved. It efficiently locates the earliest applicable map element and traverses the map until a large enough gap is found.

Finally, SetBusy reserves an identified slot in `busy_map`.

Algorithm 3: Modeling bus contention using BusyMap (continued)

```
Function FindFreeSlot (earliest,span) begin
    if busy_map.empty() then
        return earliest;
    end
    iter := busy_map.upper_bound(earliest);
    if iter = busy_map.begin() then
        gap_at := 0;
    else
        gap_at := prev(iter)→start + prev(iter)→duration;
    end
    if gap_at<earliest then
        gap_at := earliest;
    end
    while iter ≠ busy_map.end() do
        gap_duration := iter→start - gap_at;
        if span ≤ gap_duration then
            return gap_at;
        end
        gap_at := iter→start + iter→duration;
        iter++;
    end
    return gap_at;
end
Procedure SetBusy (slot,span) begin
    if busy_map.empty() then
        busy_map[slot] := span;
        return;
    end
    r := busy_map.upper_bound(slot);
    if r = busy_map.begin() then
        if r→start = slot + span then
            busy_map[slot] := span + r→duration;
            busy_map.erase(r);
        else
            // no adjacency, insert a new element
            busy_map[slot] := span;
        end
        return;
    end
    l := prev(r);
    if l→start + l→duration = slot then
        if (r ≠ busy_map.end()) and (r→start = slot + span)
          then
            // merge in between adjacent elements
            l→duration += span + r→duration;
            busy_map.erase(r);
        else
            // merge with adjacent element on the left
            l→duration += span;
        end
    else
        if (r ≠ busy_map.end()) and (r→start = slot + span)
          then
            // merge with adjacent element on the right
            busy_map[slot] := span + r→duration;
            busy_map.erase(r);
        else
            // no adjacency, insert a new element
            busy_map[slot] := span;
        end
    end
end
```

As indicated above, any adjacent elements are merged for efficiency.

V. Experiments

We have applied the proposed contention modeling approach to several SystemC model examples. We first tested two synthetic examples to validate our modeling approach's accuracy, effectiveness, and generality. Then, we evaluate the contention modeling with a real-world design, a parallel implementation of a JPEG encoder running on an RISC-V SMP model with multiple levels of caches to demonstrate the performance of the BusyMap contention modeling. The experiments are performed on an Intel Xeon(R) E-2388G host machine, running on 3.20GHz with 16 cores (8 cores with 2-way hyperthreaded).

A. Bus3Init

We first examine the accuracy of the proposed BusyMap bus on a synthetic example of three TLM-2.0 loosely-timed core initiator modules. Fig. 4a shows how cores are connected to the shared bus and memory. The three cores concurrently process a sample workload for three time units and send a transaction that occupies the bus and memory for two time units. We repeat this test pattern thrice in each core. Fig. 4b shows a sample simulation trace for the BusyUntil bus with the global quantum value of zero. As shown, the total simulated time [4] and bus contention [5] are 21 and 12 units, respectively. To validate the proposed BusyMap's temporal decoupling accuracy, we simulate the model with the BusyMap bus and sweeping global quantum values from 0 to 13 units. As shown in Table I, the simulated time and bus contention values show the high accuracy of BusyMap.

B. Hierachical buses

We also design and evaluate a TLM-2.0 LT model of cores and hierarchy of caches and interconnects to show the effectiveness of BusyMap in supporting multiple-level bus hierarchies. Fig. 5 shows a mock-up example of two cores connected to L1 caches and a high-performance bus model, such as the Advanced High-performance Bus (AHB) to shared memory. The AHB bus is connected to a peripheral bus, such as the Advanced Peripheral Bus (APB), via a bridge to slower devices like the keyboard and mouse. In contrast to the BusyUntil bus, the BusyMap bus supports multiple-level caches and buses hierarchy. Moreover, relaxing the need that components before the bus must always call `wait` to stay current with the `global_time` (condition 2 in Section III-A) would lead to built-in atomicity for transactions that the system designer can leverage for faster simulation while maintaining accurate contention modeling as we show in Section V-C.

C. Parallel JPEG encoder on RISC-V SMP virtual platform

We evaluate the simulation performance of BusyMap using a real-world parallel JPEG encoder application running on a RISC-V instruction set simulator (ISS) [15] using hierarchical caches on a symmetric multiprocessor (SMP) model (Fig. 2). Our system platform models caches as fully associative with the Least Recently Used (LRU) replacement algorithm. We model write-through with write-upgrade caches to maintain coherency between the local caches and instantly update the new value in every other local cache. We use write-through with upgrade as this coherency method ensures data consistency between the caches even when atomic operations are not guaranteed, as the `busy_until` bus does not work when multiple blocking transports are generated from the same socket (limitation due to condition 2 in Section III-A). To alleviate this issue, the cache must call `wait()` before issuing another blocking transport. During the `wait()` statement, the cache data may no longer be consistent, so write-upgrade is the best policy for this evaluation. As shown in Table II, the simulated time in the BusyMap bus matches the `busy_until` bus at lower values of global quantum. As the global quantum increases, the simulation accuracy is reduced by around 12%. However, a significant simulator run time speed-up is gained. We also measure the number of (`wait()`) statements required for local quantum keepers in cores to synchronize with the global quantum. We observe that as the global quantum value increases, the number of `wait()` calls is reduced, reducing context switching and improving simulator run time.

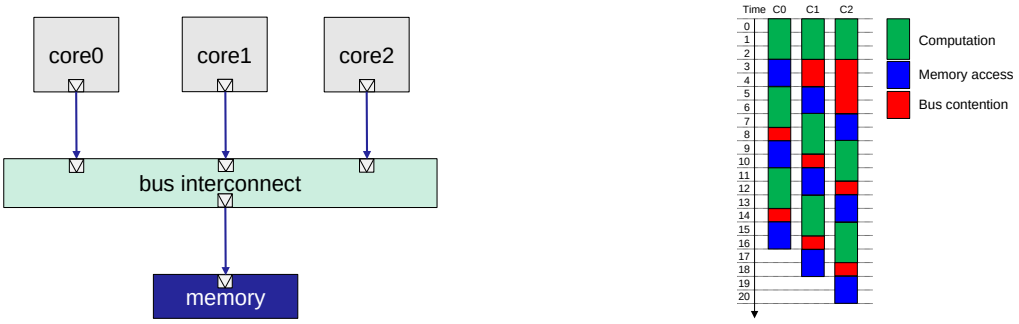
A significant advantage of the BusyMap bus is the ability to monitor contention accurately while retaining the atomicity in the model. Considering the RISC-V ISS, the core only calls the `wait()` statement after performing a blocking transport to retrieve either instructions or data from the main memory. When we introduce hierarchical caches between the core and memory into the model, these caches must be coherent with each other. If we call `wait()`, in either the caches or memory, then the atomicity of the transaction is lost, as the simulator will context switch, and another core may change the data, which could end up with incorrect data when the blocking transport returns to the core. This is a limitation when modeling more complicated write-back caches using the `busy_until` bus, as the core must wait before generating new transaction through the same socket (i.e., the cache needs to write-back data and simultaneously retrieve data from the main memory for its core). The BusyMap does not have this limitation, as it can accurately monitor bus contention and only needs to call `wait()` at the core. This enables the system designer to model more complex write-back caches to maintain coherency while guaranteeing atomicity (e.g., MESI protocol).

VI. Conclusion

Resource contention modeling is a critical aspect that must be addressed early in the design flow for effective embedded system design. However, system models that can reflect resource contention accurately must have detailed timing, which significantly slows simulation speed and is often developed later in the design stages. This paper proposes BusyMap, a novel data structure that can efficiently be used in resource contention modeling in SystemC TLM-2.0 loosely-timed and temporally decoupled models. Our experimental results show that BusyMap can successfully be used in modeling multi-level interconnects and caches in temporal decoupled loosely-

[4]The time advanced and maintained by the simulator

[5]Total accumulated contention experienced by all transactions



(a) (b)

Fig. 4: (a) TLM-2.0 model of 3 initiator modules concurrently sending transactions to a shared bus and memory (b) Simulation trace for BusyUntil bus with global quantum value of zero (simulated time = 21 units, bus contention = 12 units)

TABLE I: Total simulated time and bus contention for Bus3Init model with different global quantum values

	Simulated time		Bus contention	
Global quantum	BusyUntil	BusyMap	BusyUntil	BusyMap
0	21	21	12	12
1	21	21	12	12
2	21	21	12	12
3	21	21	12	12
4	21	21	12	12
5	21	21	12	12
6	22	20	15	14
7	22	23	15	14
8	21	25	19	14
9	21	20	19	14
10	27	20	24	14
11	27	22	24	16
12	27	27	24	16
13	29	29	32	16

TABLE II: Parallel JPEG simulation results running on RISC-V SMP

	Simulated time		Bus Contention		Simulator run-time		# wait statements (cores)		# wait statements (caches)	
Global quantum	BusyUntil	BusyMap	BusyUntil	BusyMap	BusyUntil	BusyMap	BusyUntil	BusyMap	BusyUntil	BusyMap
0ns	2.33s	2.33s	3.94s	4.00s	30m49s	18m50s	63574400	63628015	21359780	0
10ns	N/A	2.33s	N/A	4.00s	N/A	18m48s	N/A	63628015	N/A	0
100ns	N/A	2.33s	N/A	3.90s	N/A	13m33s	N/A	22157149	N/A	0
1000ns	N/A	2.44s	N/A	4.71s	N/A	11m38s	N/A	6478006	N/A	0
10000ns	N/A	2.65s	N/A	5.95s	N/A	10m20s	N/A	896267	N/A	0

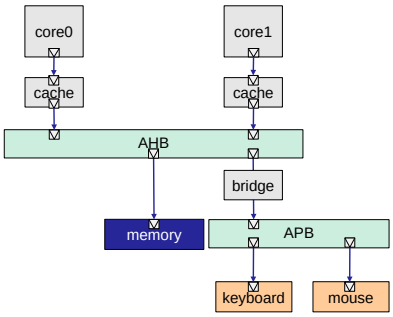


Fig. 5: Hierarchical buses with multi-level caches

timed models. The SystemC model can achieve a speedup of up to 3x on a 16-core host with only 10% accuracy loss in simulated time and bus contention. Our future work includes the evaluation of BusyMap on more industry-strength applications and improving its speed-accuracy tradeoff.

REFERENCES

[1] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the Ground Up, Second Edition*, 2nd ed. Springer Publishing Company, Incorporated, 2009.

[2] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design: Modeling, Synthesis and Verification*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[3] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[4] IEEE Computer Society, *IEEE Standard 1666-2011 for Standard SystemC Language Reference Manual*. IEEE, New York, USA, 2011.

[5] M. I. Frank, A. Agarwal, and M. K. Vernon, "Lopc: Modeling contention in parallel algorithms," p. 276–287, 1997.

[6] C. Chen and F. Lin, "An easy-to-use approach for practical bus-based system design," *IEEE Trans. Computers*, vol. 48, no. 8, pp. 780–793, 1999.

[7] A. Bobrek, J. M. Paul, and D. E. Thomas, "Shared resource access attributes for high-level contention models," in *2007 44th ACM/IEEE DAC*, 2007, pp. 720–725.

[8] D. Bertsekas and R. Gallager, *Data Networks (2nd Ed.)*. USA: Prentice-Hall, Inc., 1992.

[9] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, 2002.

[10] S. Kunzli, F. Poletti, L. Benini, and L. Thiele, "Combining simulation and formal methods for system-level performance analysis," in *Proceedings of the DATE Conference*, vol. 1, 2006, pp. 1–6.

[11] E. Wandeler, "Modular performance analysis and interface based design for embedded real time systems," Ph.D. dissertation, ETH Zurich, Zurich, Switzerland, 2006.

[12] F. Gregoretti, G. Balbo, G. Conte, and M. Marsan, "Modeling bus contention and memory interference in a multiprocessor system," *IEEE Transactions on Computers*, vol. 32, no. 01, pp. 60–72, jan 1983.

[13] A. Bobrek, J. M. Paul, and D. E. Thomas, "Stochastic contention level simulation for single-chip heterogeneous multiprocessors," *IEEE Transactions on Computers*, vol. 59, pp. 1402–1418, 2010.

[14] E. M. Arasteh and R. Dömer, "Fast loosely-timed deep neural network models with accurate memory contention," *ACM Transaction on Embedded Computer Systems*, jul 2023.

[15] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable risc-v based virtual prototype," in *2018 Forum on Specification & Design Languages (FDL)*, 2018, pp. 5–16.