Accurately Timed Transaction Level Models for Virtual Prototyping at High Abstraction Level

Kun Lu, Daniel Müller-Gritschneder and Ulf Schlichtmann
Institute for Electronic Design Automation
Technische Universität München, Munich, Germany

***Abstract*—Transaction level modeling (TLM) improves the simulation performance by raising the abstraction level. In the TLM 2.0 standard based on OSCI SystemC, a single transaction can transfer a large data block. Due to such high abstraction, a great amount of information becomes invisible and thus timing accuracy can be degraded heavily.**

We present a methodology to accurately time such block transactions and achieve high simulation performance at the same time. First, before abstraction, a profiling process is performed on an instruction set simulator (ISS). Driver functions that implement the transfer of the data blocks are simulated. Several techniques are employed to trace the exact start and end of the driver functions as well as HW usages. Thus, a profile library of those driver functions can be constructed. Then, the application programs are host-compiled and use a single transaction to transfer a data block. A strategy is presented that efficiently estimates the timing of block transactions based on the profile library. It is the first method that takes into account caching effects that influence the timing of block transactions. Moreover, it ensures overall timing accuracy when integrated in other SW timing tools for full system simulation. Experimental results show that the block transactions are accurately timed, with average error less than 1%. At the same time, the simulation gain can be up to three orders of magnitude.

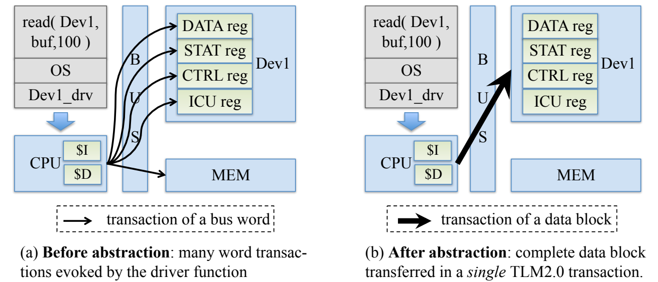
I. Introduction

Virtual prototypes (VPs) are prevalently adopted in industry for the development and verification of embedded SWs. Data communication between the HW modules in VPs can be modeled at various abstraction levels. Transaction level modeling (TLM) models the communication at high abstraction level and thus provides high simulation performance.

A. Abstraction Level of Data Flow

There have been various interpretations of abstraction levels in the domain of SoC. In the context of TLM, we consider the abstraction of the data flow. Accordingly, abstraction levels are derived from the granularity of the data flow. The term TLM can be misleading in that a *transaction* does not bind to a single abstraction level. Researchers have attempted to propose corresponding terms for transactions at different abstraction levels. Ref. [1], [2] use *packet-level TLM* or *word-level TLM* for transactions that transfer packets or bus words. Ref. [3] uses *TLM+ transaction* to refer to a transaction that transfers a data block. The TLM 2.0 standard [4] uses the term *generic payload* to incorporate the transactions at different abstraction levels. In this paper, to avoid ambiguity, we refer to a transaction of a bus word as a word transaction and a transaction of a data block as a block transaction. Correspondingly, word transactions reside at the bus word level and block transactions reside at the block level.

978-3-9810801-8-6/DATE12/© 2012 EDAA



(a) **Before abstraction**: many word transactions evoked by the driver function

(b) **After abstraction**: complete data block transferred in a *single* TLM2.0 transaction.

Fig. 1. Abstraction of the data flow.

B. CPU Model for SW Simulation

Two main paradigms exist to simulate an application program on a VP. The program can be cross compiled to the binary code of the embedded target CPU. Then an ISS interprets and executes the binary as the target CPU does. An ISS is often used as the golden reference for performance estimation, but it is prohibitively slow to simulate large programs. As a remedy for this simulation speed problem, an alternative paradigm compiles the program directly for the host CPU and executes it on the host CPU thereupon. The program is annotated with timing information for performance estimation [5], [6].

Data transfers between I/O devices and memory are implemented by driver functions [7], [8]. As illustrated in the left part of Fig. 1, to transfer a single data unit in the block, the driver function can invoke many word transactions to access memory, initialize hardware modules, as well as implement the interrupt or polling protocol. At block level, the program can use a single block transaction to transfer the whole data block (right part of Fig. 1), abstracting away the long sequence of word transactions.

At block level, it becomes difficult to time the block transactions, because a great amount of detail is missing due to abstraction. Previously, it has not been proposed how to extract sufficient timing characteristics to time the block transactions. Designers often count on empirical values for timing estimation [8], [9].

Furthermore, caching effects complicate the timing estimation for the block transactions. Two block transactions can have very different timing, even they are of the same type (e.g., *writeUart()*) and transfer the same amount of data. For the instruction cache, the surrounding codes of a driver function can cause cache conflicts and cache misses thereof. Besides, there can be cache conflicts within the driver function, e.g., if the interrupt service routine (isr) is used to transfer each unit of data in a block. For the data cache, cache misses can not be determined statically, as the data locality can only be determined dynamically during simulation. For example, assume we call a driver function to send a data buffer from memory to an I/O device. There can be many cache misses if the data block is accessed for the first time. However cache misses would not happen if we read the same block sequentially for the second time, since the data are already present in the data cache. As a result, the duration of the first driver function call can be much longer than that of the second call. Such effects of instruction and data caches, if not handled properly, can be another prominent source of error in the timing estimation of block transactions.

C. Our Methodology

We present a new methodology for the timing estimation of TLMs at block level. The methodology has two main features:

The first is a profiling process. An ISS is used to simulate only the driver functions that implement the transfer of data blocks. The entry and exit instruction addresses of those driver functions are obtained using debug information. Thus, the exact start and end of those driver functions are traced non intrusively. Further, all HW accesses during the execution of each driver function are traced. Then, the trace file is parsed to construct a profile library for the examined driver functions.

The second feature is a new timing estimation strategy for fast simulation at block level. For a particular block transaction, its timing consists of a static timing part and a dynamic timing part:

- Static timing estimation: The profile library is queried using the type (e.g., writeUart() or readCamera()) of the considered block transaction. The static timing part in the estimated duration is calculated by multiplying the average time per data unit transfer with the size of the data block.
- Dynamic timing estimation: This is the first method to handle the caching effects of both the instruction and data caches. The instruction cache timing behavior is simulated using the address space stored in a block transaction's profile. The timing behavior of the data cache is simulated by applying data flow analysis to obtain the address of data memory accesses.

With the presented methodology, timing at block level is estimated with high accuracy. Experimental results show an average timing estimation error less than 0.5%, while the simulation gain can be up to three orders of magnitude. The remainder of this paper is organized as follows: Section II describes related work. Section III and IV present the two main steps in timing estimation of block transactions. Section V shows the efficacy of the proposed method with experimental results. Section VI concludes this paper.

II. Related work

In the simulation of embedded SWs, timing simulation has been considered by researchers to be orthogonal to the functional simulation. Accordingly, the simulation can be performed untimed (programmer's view) or timed (programmer's view with time) [10]. As for the timing of transactions, loosely timed or approximately timed styles have been proposed by [4]. Up to now, a majority of previous approaches aim to time the transactions at word level [11], [12].Efficient methods to time block transactions have not been proposed. For example, authors in [8] use time-consuming heuristic manual fitting to annotate timing of block transactions. Ref. [9] uses an approximate estimation to time the block transactions. Both of them do not consider the complex caching effects described in Sec. I.

Eventually, timed block transactions should be integrated into SW timing annotation tools ([5], [6]) to enable full system simulation. Currently, these tools focus on timing of a program without involving OS or I/O devices (referred to as bare metal mode [13]). When integrated together, the interaction between the block transactions and the rest of the program affect the timing of each other (e.g., they have instruction cache conflicts). To ensure timing accuracy in full system simulation, proper measures should be taken in the timing estimation of block transactions.

III. Tracing and Profiling Tool Chain

The profiling process extracts the timing characteristics for the driver functions in ISS simulation. A profile function, which is independent of the application program, is used to evoke the driver functions of different types to transfer data blocks. Tracing of the driver functions is given in Sec. III-A. The construction of the profile function and the extracted profiles are explained in Sec. III-B.

A. Tracing mechanism

The tool first extracts the entry and exit instruction addresses of all the functions with the help of debuggers, as shown in the left part of Fig. 2. Those addresses are noted as the instruction space boundaries (*isb*). Then a SW monitor is added to the ISS. At the start of simulation, the SW monitor reads in the *isb*s of all the functions. During simulation, the ISS provides the following information to the SW monitor for each binary instruction it simulates: the instruction code and the address of this instruction. If the SW monitor detects that the instruction enters or exits a function (e.g. *jal* or *jr* for a MIPS CPU), then it checks the address against those extracted *isb*s. In this way, the start and end of the functions, including the driver functions, can be traced exactly. In addition, HW activities are also traced, including the access of a module, cache misses, etc. As a result, tracefiles and waveforms are generated.

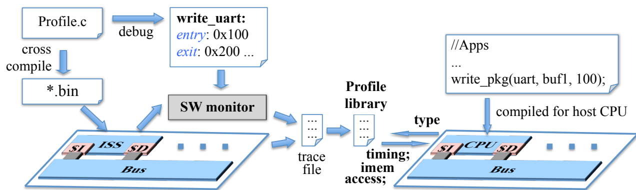


Fig. 2. Fully automated tool chain.

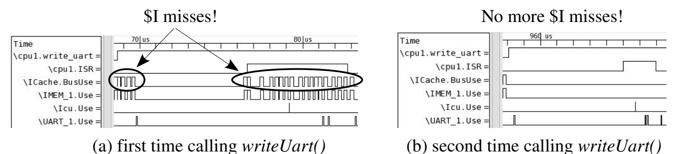


Fig. 3. Waveforms of writeUart().

TABLE I
SAMPLE PROFILE FOR A DRIVER FUNCTION *writeUart()*

Type	writeUart
Ave. duration per data unit	8.8 us
Instr. address space	0x3020, 0x3040, 0x3060, 0x20, ...

B. *The profile library*

In the profile function, a driver function is called twice consecutively to transfer the same block. For the second call, the driver function's instructions and the transferred data block are already loaded in the caches, thus dynamic caching effects are excluded. So, traces of the second call are parsed to extract the profiles for the driver functions of each type. The profile includes statically determined timing parameters, instruction memory accesses and other HW usages. A sample profile for the driver function *writeUart()* is given in Tab. I. It provides the following information:

1) The average duration to transfer one data unit of a block (a char in this case) is 8.8 us. The average duration is calculated by dividing the duration of the second driver function by the block size.
2) Its instruction address space is constituted by a sequence of addresses: 0x3020, 0x3040, etc. Here, an address is truncated to a multiple of the size of a cache line (32 bytes). Thus, 0x3020 represents an address range from 0x3020 to 0x303f. If an instruction cache miss occurs due to accessing the instruction at 0x3028, 8 instructions (4 bytes each) starting from 0x3020 will be transferred to the instruction cache from memory. The reason of using such cache-line-aligned addresses is that only a non-functional cache timing simulation will performed (e.g., [5]). Hence, only the bits related to tag and cache set in an address are significant. The bits to address a byte in a cache line can be ignored. During the execution of this driver function, the accessed instructions and data are loaded to the caches, resulting in dynamic timing to be analyzed in Sec. IV-B.

IV. SIMULATION AND DYNAMIC TIMING ESTIMATION

At this stage, the application program is simulated using host compilation. One block transaction is used to transfer a complete data block, based on the OSCI TLM 2.0 standard. When a block transaction is initiated, its type is used to query the profile library. Timing estimation is performed with the profile of this block transaction. This process is illustrated in the right part of Fig. 2. The estimated duration of a block transaction is the sum of a static timing part and a dynamic timing part, as in:

$$T_{block} = T_{static} + T_{dyn} \tag{1}$$

The details of their derivations are given in the following.

A. *Static timing*

This static part in the estimated duration of a block transaction is calculated by multiplying the block size with the average duration (T_{ave}) to transfer one data unit in the block.

$$T_{static} = size \cdot T_{ave} \tag{2}$$

One special issue needs to be made clear. When simulated on ISS, the driver function can call the interrupt service routine (*isr*) for every transferred data unit. It is possible that the instructions of the *isr* compete on certain instruction cachelines with the instructions of the driver function. Cache misses due to such cache conflicts will be observed for every transferred data unit. Since such penalty is internal to the driver function, it is included in the profiled average duration. Hence, the calculated static timing part in Equ. 3 can reflect the timing behavior when internal instruction cache conflicts exist.

B. *Dynamic timing*

This dynamic part is to tackle dynamic caching effects caused by both instruction and data caches in estimating the duration of a block transaction. Accordingly, it consists of two main components:

$$T_{dyn} = \Delta T_I + \Delta T_D \tag{3}$$

ΔT_I and ΔT_D are the miss penalty of instruction and data caches respectively. In the following, we show how to compute these two components.

1) Timing of instruction cache accesses: There are two observations regarding the instruction cache timing estimation.

Firstly, when a driver function is executed for the first time, it accesses the instructions in its instruction space, which are not yet in the instruction cache. Thus, the instruction cache can experience many cache misses and it will access the bus many times to load the required instructions from memory. However, subsequent calls to this driver function will not experience those cache misses, since the instructions have been loaded. An example is given in Fig. 3. *CPU1.write_uart*=1 means the the function *writeUart()* is in execution. *ICache.busUse*=1 means the instruction cache is transfering data from memory over bus. Fig. 3(a) is the waveform when *writeUart()* is called for the first time. Many instruction cache misses occur during

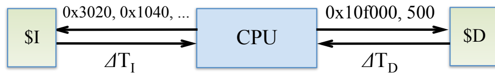
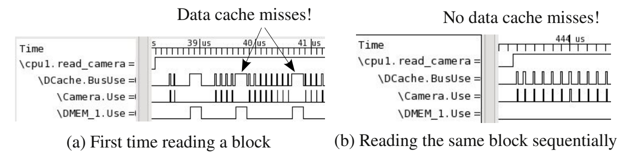


Fig. 4. New cache access modes for block transactions.



(a) First time reading a block (b) Reading the same block sequentially

Fig. 5. Dynamics of data cache misses.

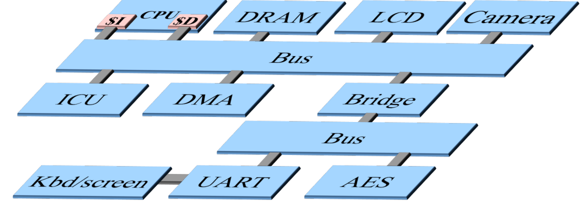


Fig. 6. HW architecture.

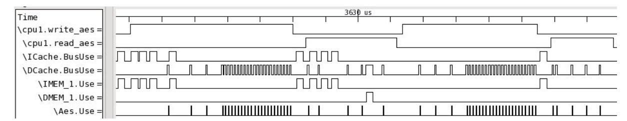


Fig. 7. Traced driver functions of *writeAes* and *readAes*.

the execution of *writeUart()*. The instruction cache accesses bus to load the required instructions, as shown by the trace of *ICache.BusUse*. In contrast, Fig. 3(b) is the trace when *writeUart()* is called for the second time subsequently. No instruction cache misses occur.

Secondly, the surrounding codes of a driver function might replace the driver function's instructions in the instruction cache and introduce additional cache misses. Consider the code in Lis. 1:

```
1:    ...
2:    write(dev1, buf1, strlen(buf1));
3:    ...
```

Listing 1. Example of cache conflicts external to a driver function

At line 2, instructions of the function *strlen()* might compete with the instructions of the driver function *write(dev1,...)*, which affects the timing of the latter. This part of timing can not be predicted statically, since the driver functions can be called in various code contexts and it can not be determined in advance whether their instructions are in cache or not.

These issues are handled as follows in our timing estimation strategy. When a block transaction starts, the cache line aligned instruction addresses in its instruction address space are obtained from the profile library. They are used to simulate the timing behavior of the instruction cache. To this end, a new access mode is added to the cache, which is exemplified in the left part of Fig. 4. In this mode, the cache accepts a list of addresses as input, and calculates an overall miss penalty (ΔT_I) for this address list as output. For this, the tags in the instruction cache are simulated. This is important when timing estimation of block transactions is integrated with SW timing annotation tools (e.g. [5]), since both perform instruction cache timing simulation.

2) Timing of data cache accesses: When reading/writing a data block from/to an I/O device, data in the block are sequentially stored to or loaded from memory. With data cache, this process can have undeterministic cache misses and bus accesses. Consider the case in Lis. 2.

```
1:    ...
2:    read(dev2, buf2, 100);
3:    read(dev2, buf2, 100);
4:    ...
```

Listing 2. Example of data cache dynamics

At line 2, the program reads data from device dev2 and stores them in buf2. Assuming buf2 is used for the first time, there will be a cache miss when storing the data to buf2[0]. In a write-back cache, the victim cacheline will firstly be written back to memory if it is dirty. Then, assuming 32-byte cachelines and 32-bit bus width (thus 4 bytes per word), 8 words will be transferred to cache from memory. Suppose buf2[0] locates at the first position of the freshly loaded cacheline, then the following accesses of buf2[1] until buf2[7] will not yield cache misses. Similarly, cache misses will occur when reading data into buf2[8], buf2[16], etc. In contrast, read to buf2 in line 3 will not give data cache misses since buf2 has already been placed in the data cache. Fig. 5 exemplifies such data cache dynamics.

To estimate such dynamic data cache behavior, we first obtain the address and the size of the data block for data cache simulation when a block transaction starts (see Fig. 4). In this access mode, the data cache chops the address range of the block by the size of cachelines. Then, starting from the given address, it will sequentially access those cachelines and simulate the cache misses. To obtain the address of a block, we use the value of an annotated stack pointer (*sp*) variable, since it gives the exact address of local variables. The variable *sp* is decreased/increased by a value when a function is entered/exited. This value is the size of the function's local stack. Consider the example in Lis. 3: the address for the variable *block* can be expressed as *sp + 20*.

```
1: void func1(){       //assume a local stack of 200 bytes
2:  char  var[20];     //locate within [sp,sp+20)
3:  char  block[100];  //locate within [sp+20,sp+120)
4:   ...
5: }
```

Listing 3. Sample code illustrating data address annotation

V. Experimental Results and Analysis

In the following, profiling results based on the method shown in Sec. III are given in Sec. V-A. Timing estimation results based on the method discussed in Sec. IV are given in Sec. V-B and V-C.

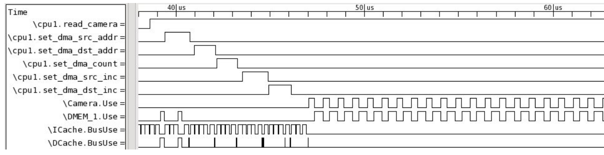


Fig. 8. Traced driver functions of readCamera using DMAC.

A. Set-up and profiling results

The set-up of the HW architecture is shown in Fig. 6.

The application program can initiate the following communication between I/O devices and memory:

- Write 16 bytes to AES for encryption
- Read the encrypted data back from AES
- Write an array of characters to the UART
- Read a frame from the camera, performed by CPU.
- Read a frame from the camera, performed by DMAC.
- Write a frame to the LCD, performed by CPU.
- Write a frame to the LCD, performed by DMAC.

In the profiling process, a program calls the driver functions for the above mentioned data transfers. This program is simulated by an ISS. Applying the technique described in Sec. III, the trace file, waveforms and profile library are obtained. It took around 3 minutes for the tool chain to complete the whole profiling process. The waveform for *writeUart()* is already given in Fig. 3. The waveform for *writeAes()* and *readAes()* is given in Fig. 7. In contrast to *writeUart()*, AES encryption is much faster than the UART transmission. Thus, instead of using the *isr*, polling is used to wait for the encryption to be complete. So many accesses to the AES's status register are observed in the waveform. The waveform for *readCamera()* without using the DMAC is already given in Fig. 5. Using the DMAC, the waveform is in Fig. 8. The waveforms for *writeLcd()* are similar to those for *readCamera()*.

B. Communication centric scenarios

The purpose of the following communication centric experiments is to examine two aspects of the timed block transactions: 1. How much performance gain can they contribute alone; 2. How accurate is the timing estimation.

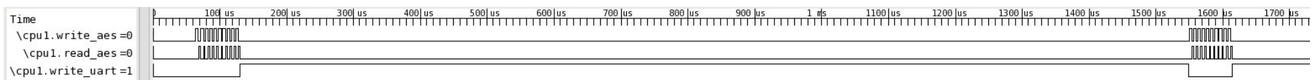
1) Application 1: In this case, the application program sends 4 words (16 bytes) from buffer 1 to AES for encryption and then reads the encrypted data back into buffer 2. It continues until buffer 1 is encrypted. Then it sends buffer 2 to the UART for display. This process is repeated for many rounds. The program is simulated at word level on an ISS and at block level using host compilation, respectively. Waveforms in Fig. 9 show the traced start and end of the driver functions and block transactions for the first round. The timing discrepancy at the beginning is due to booting the system, which is not timed when the program is host compiled. The timing of the block transactions matches closely to that in the ISS simulation. Quantitatively, the average timing estimation error of the block transactions is less than 1%. As for performance, a gain of $700\times$ is measured. This gain can be further increased by $5\times$ if the dynamic cache timing is not simulated. But the timing estimation error is then enlarged to 21%. The cache simulation overhead is large because our cache model is implemented as a HW module in SystemC. Besides, the SW containing cache accesses is host-compiled into a static library, which is then compiled with the HW system in SystemC. Thus, the cache accesses in the SW are dynamically linked in simulation and cause large overhead. This can be improved in the future by using a SW cache model. Another limit of the gain is that there are 2 context switches for each block transaction induced by (1) one *wait* to perform the TLM 2.0 transaction and (2) another transaction to check the status register of AES or UART after the block transaction completes.

2) Application 2: In this case, the program first reads two frames (4K words each) consecutively from the camera into two buffers. Then it writes the buffers to the LCD module. Such operations are repeated several times. Firstly, without using DMAC, waveforms are given in Fig. 10, Please note that, when reading the frame the first time, the duration of the driver function is around 25% longer than that for the second time. This is caused by the data cache misses, as described in Sec. IV-B. This timing behavior is very well estimated for the timed block transactions (see Fig.10(b)). The average timing estimation error for the block transactions is less than 0.5%. The performance gain is around $1200\times$. This gain is high, since the block size is large. This gain can be further increased by $4\times$ if the dynamic cache timing simulation is disabled, at the expense of an increased timing estimation error.

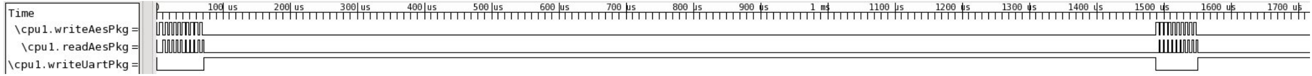
Then the test is performed with DMAC enabled. Now, to read the camera, the ISS only needs to configure the DMAC (see Fig. 8). Then the DMAC initiates transactions with burst length of 32 to establish high speed transfer from the camera. Thus, the simulation time is substantially shorter for the ISS. Consequently, the measured gain of block based simulation is lowered to only around $90\times$. The timing estimation accuracy is still high, with an average error less than 0.5%.

C. Computation centric scenario

In this test, the timing estimation of block transactions is integrated into a SW timing annotation tool. The application program reads a frame from the camera. Then it performs an Rgb to YCbCr conversion algorithm and mixes the converted YCbCr frame with another YCbCr frame. When the program is host compiled, timing information is annotated in the conversion and mixing algorithm. The block transactions, which read frames from the camera, are timed by the proposed method. As is shown in Fig. 11, the timing of block transactions matches closely the timing in ISS based simulation. In the worst case, the estimated duration of a block transaction is 6% shorter. This is because the applied SW timing tool does not perform data memory annotation, which can cause data cache conflicts and hence prolong a block transaction. The gain in this case is close to $2000\times$. One reason for this high gain is a fast cache simulation strategy in the annotated application program. This strategy symbolizes the cache simulation of a



(a) Results when the program is executed on ISS.

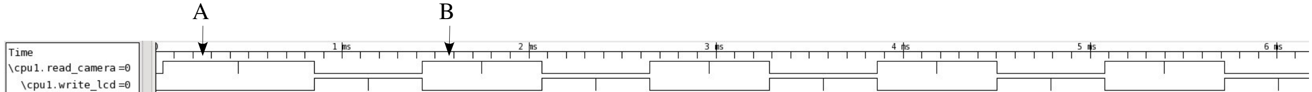


(b) Results when the program is host compiled and timed at block level.

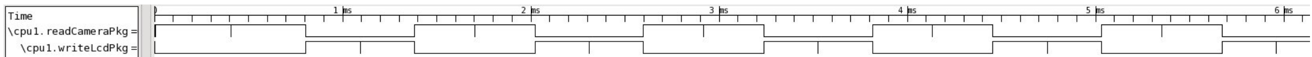
Fig. 9. Timing comparison for application 1.

A is around 25% longer than B, though they transfer the same amount of data

A B

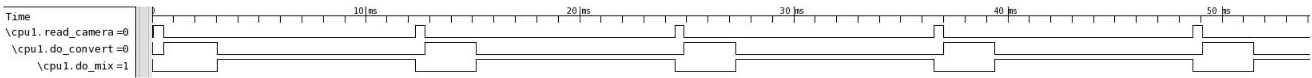


(a) Results when the program is executed on ISS.

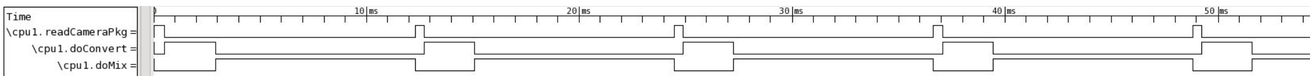


(b) Results when the program is host compiled and timed at block level.

Fig. 10. Timing comparison for application 2 without using DMAC.



(a) Results when the program is executed on ISS.



(b) Results when the program is host compiled and timed at block level.

Fig. 11. Timing comparison after integration with a SW timing tool.

loop body into a function and moves it outside of the loop body. This function uses the loop iteration count as a parameter to estimate the cache misses. Therefore, the overhead of the cache simulation is very little for a loop with large iteration counts, such as the case of the examined program.

VI. CONCLUSION

This paper proposes a methodology to time block transactions for TLM based virtual prototypes. First, a tracing and profiling process is automated to contruct a profile library for the block transactions. Then, a timing estimation strategy is employed to time the block transactions with their profiles. Various caching effects at both the instruction and data caches are handled effectively by the timing estimation strategy. Experiments show that the block transactions are accurately timed (average error less than 1%). At the same time, the simulation gain can be up to three orders of magnitude.

ACKNOWLEDGEMENTS

This work has been supported by the German Federal Ministry of Education and Science (BMBF) in the SANITAS project under grant nr. 01 M 3088.

REFERENCES

[1] A. Wieferink, M. Doerper, T. Kogel, et al., and H. Meyr, "Early ISS Integration into Network-on-Chip Designs," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2004.

[2] T. Kogel, R. Leupers, and H. Meyr, *Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms*. Springer, 2006.

[3] W. Ecker, S. Heinen, and M. Velten, "Using a dataflow abstracted virtual prototype for HdS-design," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2009, pp. 293–300.

[4] OSCI, *OSCI TLM-2.0 Language Reference Manual*, 2009.

[5] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," in *ACM/IEEE Design Automation Conference (DAC)*, 2008.

[6] D. Mueller-Gritschneder, K. Lu, and U. Schlichtmann, "Control-flow-driven Source Level Timing Annotation for Embedded Software Models on Transaction Level," in *EUROMICRO Conference on Digital System Design (DSD)*, Sep. 2011.

[7] P. Gerin, X. Guerin, and F. Petrot, "Efcient Implementation of Native Software Simulation for MPSoC," in *Design, Automation and Test in Europe (DATE)*, 2008.

[8] W. Ecker, V. Esen, and M. Velten, "TLM+ modeling of embedded HW/SW systems," in *Design, Automation and Test in Europe (DATE)*, 2010.

[9] M. Ariyamparambath, D. Bussaglia, B. Reinkemeier, T. Kogel, and T. Kempf, "A Highly Efficient Modeling Style for Heterogeneous Bus Architectures," in *IEEE International Symposium on System-on-Chip*, 2003.

[10] A. Donlin, "Transaction Level Modeling: Flows and Use Models," in *CODES ISSS*, 2004.

[11] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and Accurate Resource Conflict Simulation for Performance Analysis of Multi-Core Systems," in *Design, Automation and Test in Europe (DATE)*, 2011.

[12] E. Cheung, H. Hsieh, and F. Balarin, "Memory subsystem simulation in software TLM/T models," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2009.

[13] O. Almer, I. Boehm, T. Edler, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham, "Scalable Multi-Core Simulation Using Parallel Dynamic Binary Translation," in *International Conference on Systems, Architectures, Modeling and Simulation (SAMOS)*, 2011.