An On-Chip AHB Bus Tracer With Real-Time Compression and Dynamic Multiresolution Supports for SoC

Fu-Ching Yang, Member, IEEE, Yi-Ting Lin, Chung-Fu Kao, and Ing-Jer Huang, Member, IEEE

Abstract—This paper proposes a multiresolution AHB on-chip bus tracer named SYS-HMRBT (AHB multiresolution bus tracer) for versatile system-on-chip (SoC) debugging and monitoring. The bus tracer is capable of capturing the bus trace with different resolutions, all with efficient built-in compression mechanisms, to meet a diverse range of needs. In addition, it allows users to switch the trace resolution dynamically so that appropriate resolution levels can be applied to different segments of the trace. On the other hand, SYS-HMRBT supports tracing after/before an event triggering, named post-triggering trace/pre-triggering trace, respectively. SYS-HMRBT runs at 500 MHz and costs 42 K gates in TSMC 0.13-μm technology, indicating that it is capable of real time tracing and is very small in modern SoCs. Experiments show that the bus tracer achieves very good compression ratios of 79%–96%, depending on the selected resolution mode. As a case study, it has been integrated into a 3-D graphics SoC to facilitate the debugging and monitoring of the system behaviors. The SoC has been successfully verified both in field-programmable gate array and a test chip.

Index Terms—AHB, AMBA, compression, multiresolution, periodical triggering, post-T trace, pre-T trace, real time trace, system-on-chip (SoC) debugging.

I. INTRODUCTION

The on-chip bus is an important system-on-chip (SoC) infrastructure that connects major hardware components. Monitoring the on-chip bus signals is crucial to the SoC debugging and performance analysis/optimization. Unfortunately, such signals are difficult to observe since they are deeply embedded in a SoC and there are often no sufficient I/O pins to access these signals. Therefore, a straightforward approach is to embed a bus tracer in SoC to capture the bus signal trace and store the trace in an on-chip storage such as the trace memory which could then be off loaded to outside world (the trace analyzer software) for analysis.

Unfortunately, the size of the bus trace grows rapidly. For example, to capture AMBA AHB 2.0 [1] bus signals running at 200 MHz, the trace grows at 2 to 3 GB/s. Therefore, it is highly desirable to compress the trace on the fly in order to reduce the trace size. However, simply capturing/compressing bus signals is not sufficient for SoC debugging and analysis, since the debugging/analysis needs are versatile: some designers need all signals at cycle-level, while some others only care about the transactions. For the latter case, tracing all signals at cycle-level wastes a lot of trace memory. Thus, there must be a way to capture traces at different abstraction levels based on the specific debugging/analysis need.

This paper presents a real-time multi-resolution AHB on-chip bus tracer, named SYS-HMRBT (AHB multiresolution bus tracer). The bus tracer adopts three trace compression mechanisms to achieve high trace compression ratio. It supports multiresolution tracing by capturing traces at different timing and signal abstraction levels. In addition, it provides the dynamic mode change feature to allow users to switch the resolution on-the-fly for different portions of the trace to match specific debugging/analysis needs. Given a trace memory of fixed size, the user can trade off between the granularity and trace length to make the most use of the trace memory. In addition, the bus tracer is capable of tracing signals before/after the event triggering, named pre-T/post-T tracing, respectively. This feature provides a more flexible tracing to focus on the interesting points.

This feature provides a more flexible tracing to focus on the interesting points.

The rest of this paper is organized as follows. Section II surveys the related work. Section III discusses the features in trace granularity and trace direction. Section IV presents the hardware architecture of our bus tracer. Section V provides experiments to analyze the compression ratio, trace depth, and cost of our bus tracer. A case study is also conducted to integrate the bus tracer with a 3-D graphics SoC. Finally, Section VI concludes this paper and gives directions for future research.

II. RELATED WORK

Since the huge trace size limits the trace depth in a trace memory, there are hardware approaches to compress the traces. The approaches can be divided into lossy and lossless trace compression.

The lossy trace compression approach achieves high compression ratio by sacrificing the accuracy; the original signals cannot be reconstructed from the trace. The purpose of this approach is to identify if a problem occurs. Anis and Nicolic [2] use the multiple input signature register (MISR) to perform lossy compression. The results are stored in a trace memory and compared with the golden patterns to locate the range of the erroneous signals. The locating needs rerunning the system several times with finer and finer resolution until the size of the search range can fit in the trace memory. Such approach

The approaches can be divided into lossy and lossless trace compression.

The lossy trace compression approach achieves high compression ratio by sacrificing the accuracy; the original signals cannot be reconstructed from the trace. The purpose of this approach is to identify if a problem occurs. Anis and Nicolic [2] use the multiple input signature register (MISR) to perform lossy compression. The results are stored in a trace memory and compared with the golden patterns to locate the range of the erroneous signals. The locating needs rerunning the system several times with finer and finer resolution until the size of the search range can fit in the trace memory. Such approach...
is suitable for deterministic and repeatable system behaviors. However, for a complex SoC with multiple independent IPs, the on-chip bus activities are usually not deterministic and repeatable. Therefore, lossless compression approaches are more appropriate for real-time on-chip bus tracing.

Existing on-chip bus tracers mostly adopt lossless compression approaches. ARM provides the AMBA AHB trace macrocell (HTM) [3] that is capable of tracing AHB bus signals, including the instruction address, data address, and control signals. The instruction address and control signals are compressed with a slice compression approach (to be explained shortly). On the other hand, the data address is recorded by simply removing the leading zeros. The HTM supports a limited level of trace abstraction by removing bus signals that are in IDLE or BUSY state. The AMBA navigator [4] traces all AHB bus signals without compression. In the bus transfer mode, it also has a limited level of trace abstraction by removing bus signals which are in IDLE, BUSY, or non-ready state. The AHBTRACE in GRLIB IP library [5] captures the AMBA AHB signals in the uncompressed form. In addition, it does not have trace abstraction ability.

There are many research works related to the bus signal compression. We characterize the bus signals into three categories: program address, data address/data, and control signals. We then review appropriate compression techniques for each category. For program addresses, since they are mostly sequential, a straightforward way is to discard the continuous instruction addresses and retain only the discontinuous ones, so called branch/target filtering. This approach has been used in some commercial tracers, such as the TC1775 trace module in TriCore [6] and ARM’s Embedded Trace Macrocell (ETM) [7]. The hardware overhead of these works is usually small since the filtering mechanism is simple to be implemented in hardware. The effectiveness of these techniques, however, is mainly limited by the average basic block size, which is roughly around four or five instructions per basic block [7], [8]. Other technique such as the slice compression approach [3] targets at the spatial locality of the program address. This approach partitions a binary data into several slices and then records all the slices of the first data and then only part of the slices of the succeeding data that are different from the corresponding slices of the previous one (usually the lower bit positions of the data). For data address/value, the most popular method is the differential approach which records the difference between consecutive data. Since the difference usually could be represented with less number of bits than the original value, the information size is reduced. Hopkins and McDonald–Maier showed that the differential method can reduce the data address and the data value by about 40% and 14%, respectively [9]. For control signals, ARM HTM [3] encodes them with the slice compression approach: the control signal is recorded only when the value changes.

As mentioned, compressing all signals at the cycle-accurate-level does not always meet the debugging needs. As SoCs become more complex, the transaction-level debugging becomes increasingly important, since it helps designers focus on the functional behaviors, instead of interpreting complex signals. Tabbara and Hashmi [10] propose the transaction-level SoC modeling and debugging method. The proposed transactors, attaching to the on-chip bus, recognize/monitor signals and abstract the signals into transactions. The transactions, bridging the gap between algorithm-level and the signal-level, enable easy design exploration/debugging/monitoring. Vermeulen et al. [11] propose a communication-centric intrusive debugging method based on the transaction level. They point out that the traditional hardware and software debugging cannot work collaboratively, since the software debugging is at the functional level and the hardware debugging is at the signal level. As a solution, the transaction-level debugging can provide software and hardware designers a common abstraction level to diagnose bugs collaboratively, and thus, help focus problems quickly. Both works indicate that the transaction-level debugging is a must in SoC development.

Motivated by the related works, our bus tracer combines abstraction and compression techniques in a more aggressive way. The goal is to provide better compression quality and multiple resolution traces to meet the complex SoC debugging needs. For example, our bus tracer can provides traces at cycle-level and transaction-level to support versatile debugging needs. Besides, features such as the dynamic mode change and bidirectional traces are also introduced to enhance the debugging flexibility.

### III. Trace Granularity and Trace Direction

The multiresolution trace mode and the pre/post-T tracing are two important features for effective SoC debugging and monitoring. They are discussed in this section in terms of trace granularity and trace direction.

#### A. Trace Granularity—Multiresolution Trace

This section first introduces the definitions of the abstraction level. Then, it discusses the application for each abstraction mode.

1) **Timing and Signal Abstraction Definition:** The abstraction level is in two dimensions: timing abstraction and signal abstraction.

At the timing dimension, it has two abstraction levels, which are the cycle level and transaction level, as shown in Table I. The cycle level captures the signals at every cycle. The transaction level records the signals only when their value change (event triggering). For example, since the bus read/write control signals do not change during a successful transfer, the tracer only records this signal at the first and last cycles of that transfer. However, if the signal changes its value cycle-by-cycle, the transaction-level trace is similar to the cycle-level trace.

At the signal dimension, first, we group the AHB bus signals into four categories: program address, data address/value, access control signals (ACS), and protocol control signals (PCS). Then, we define three abstraction levels for those signals. As shown in Table II, they are full signal level, bus state level, and

<table>
<thead>
<tr>
<th>Timing Abstraction</th>
<th>Cycle level</th>
<th>Transaction level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time granularity</td>
<td>Cycle accurate</td>
<td>Event triggering</td>
</tr>
</tbody>
</table>

| TABLE I

<table>
<thead>
<tr>
<th>Cycle level</th>
<th>Transaction level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time granularity</td>
<td>Cycle accurate</td>
</tr>
</tbody>
</table>
The full signal level captures all bus signals. The bus state level further abstracts the PCS by encoding them as states according to the bus-state-machine (BSM) shown in Fig. 1. The states represent the bus handshaking activities within a bus transaction. The master state level further abstracts the bus state level by only recording the transfer activities of bus masters and ignoring the handshaking activities within transactions. This level also ignores the signals when the bus state is IDLE, WAIT, and BUSY.

The BSM is designed based on the AMBA AHB 2.0 protocol to represent the key bus handshaking activities within a transaction. The transitions between BSM states follow the AMBA protocol control signals. For example, in the beginning (state 0), if the master is granted the bus (HGRANT = true), it enters start state (state 1). After, the master begins to transfer by first acknowledging the transfer type, which is a sequential transfer (HTRANS = SEQ) or a nonsequential transfer (HTRANS = NONSEQ). If it is a successful transfer, the BSM goes to the normal state (state 2). After it is in state 2, if the slave is busy, the BSM enters to the wait states (HREADY = false and HRESP is OK). Later on, if the slave can finish the transfer, the BSM changes from state 3 to state 4. (HREADY = true and HRESP indicates OK).

Combining the abstraction levels in the timing dimension and the signal dimension, we provide five modes in different granularities, as Fig. 2 shows. They are Mode FC (full signal, cycle level), Mode FT (full signal, transaction level), Mode BC (bus state, cycle level), Mode BT (bus state, transaction level), and Mode MT (master state, transaction level). We will discuss the usage of each mode in the following.

2) Applications of Abstraction Modes: At Mode FC, the tracer traces all bus signals cycle-by-cycle so that designers can observe the most detailed bus activities. This mode is very useful to diagnose the cause of error by looking at the detail signals. However, since the traced data size of this mode is huge, the trace depth is the shortest among the five modes. Fortunately, it is acceptable since designers using the cycle-level mode trace only focus on a short critical period.

At Mode FT, the tracer traces all signals only when their values are changed. In other words, this mode traces the untimed data transaction on the bus. Comparing to Mode FC, the timing granularity is abstracted. It is useful when designers want to skim the behaviors of all signals instead of looking at them cycle-by-cycle. Another benefit of this mode is that the space can be saved without losing meaningful information. Thus, the trace depth increases.

At Mode BC, the tracer uses the BSM, such as NORMAL, IDLE, ERROR, and so on, to represent bus transfer activities in cycle accurate level. Comparing to Mode FC, although this mode still captures the signals cycle-by-cycle, the signal granularity is abstracted. Thus, designers can observe the bus handshaking states without analyzing the detail signals. The benefit is that designers can still observe bus states cycle-by-cycle to analyze the system performance.

At Mode BT, the tracer uses bus state to represent bus transfer activities in transaction level. The traced data is abstracted in both timing level and signal level; it is a combination of Mode BC and Mode BT. In this mode, designers can easily understand the bus transactions without analyzing the signals at cycle level.

At Mode MT, the tracer only records the master behaviors, such as read, write, or burst transfer. It is the highest abstraction level. This feature is very suitable for analyzing the masters’ transactions. The major difference compared with Mode BT is that this mode does not record the transfer handshaking activities and does not capture signals when the bus state is IDLE, WAIT, and BUSY. Thus, designers can focus on only the masters’ transactions. Please note that there is no mode supporting master operation trace at cycle level, since the intention of observing master behaviors is to realize the whole picture. Tracing master behaviors at cycle level is meaningless and can be replaced with Mode BC.
Multiresolution trace has two advantages for efficient SoC debugging. First, it provides the customized trace for diverse debugging purposes. Depending on the debugging purpose, designers can select a preferred abstraction level to observe bus signal variation. For designers debugging at a higher abstraction level, it saves a lot of time analyzing the skeleton of system operations. The idea is to make the hardware debugging process similar to the software debugging process. Designers can use the higher abstraction level trace to obtain the top view and then switch to the lower abstraction level trace on-the-fly to check the detail signals. Second, the multiresolution tracing saves trace sizes. Since higher-abstraction-level traces capture abstracted data, the required space is smaller. Therefore, given a fixed-size trace memory, the trace depth (cycle) in the higher abstraction level is larger than the traces in the lower abstraction level.

3) Dynamic Mode Change: Our bus tracer also supports dynamic mode change (DMC) feature. This feature allows designers to change the trace mode dynamically in real-time. As Fig. 3 shows, the trace mode changes seamlessly during execution.

Dynamic mode change has two benefits. One is that it provides customized traces according to the debugging purpose. The other is that designers can make tradeoffs between the trace granularity and the trace depth. Thus, the trace memory utilization is more efficient. Fig. 3 shows an example using dynamic mode change to diagnose a suspected bug. At first, designers can use Mode MT to have the top view so that they can skim the master behaviors very quickly. Then, when the time is closed to the suspected bug, they can switch to Mode BT. This provides more information about all operations on the bus and thus, designers can check the detail operations. It also helps establishing the time line of system behaviors. After, designers switch to the Mode FC to focus on every signal at cycle level for error diagnosis. Please note although the trace size per cycle is huge in this mode, it is usually not necessary to trace a long period. Finally, after Mode FC, designers can switch to Mode BC to see what operations are affected by this bug. Since the behavior at every cycle is worth noticing, this mode preserves the cycle-level trace. However, since designers only care about the behaviors instead of all signals, this mode abstracts the signal level and speeds up the debugging process.

The dynamic mode change is achieved by setting up the event registers. The event registers define the start/stop time of a trace and the trace mode. Thus, when the trigger condition meets and a new trace begins, the new trace starts in the trace mode specified in the event registers. Details are discussed in Section IV.

To provide better debugging flexibility, the captured traces can be abstracted into higher abstraction level traces by our trace analyzer software. For example, the traces of mode FC can be abstracted into traces of mode FT, mode BC, mode BT, and mode MT. The traces of mode BC can be abstracted into traces of mode BT and mode MT. This feature increases the debugging flexibility since designers can understand the waveform more quickly in higher abstraction level and narrow down the debugging range in the lower-abstraction-level waveform.

B. Trace Direction: Pre-T/Post-T Trace

Supporting both trace directions provides the flexible debugging strategies. As Fig. 4 shows, the post-T trace captures signals after a triggering event, while the pre-T trace captures signals before the triggering event. The post-T trace is usually used to observe signals after a known event. The pre-T trace is useful for diagnosing the causes of unexpected errors by capturing the signals before the errors.

The mechanisms of the pre-T trace and the post-T trace are different. The Post-T trace is simpler since the start time and the stop time are known. It is activated when the target event is matched and is turned off when the trace buffer is full. On the other hand, the stop time of the pre-T trace is unpredictable. The solution is to start tracing as soon as system reset (or some other turning-on event). When the trace buffer is full, the new trace data wrap around the trace buffer, which means the oldest data are sacrificed for the newest ones.

Wrapping around the trace buffer causes a problem when the trace needs to be compressed. Typical lossless compression algorithms work by storing some initial (previous) states of the trace first and then calculate the relationship between the current data and the previous states. Since the size of the relationship is smaller than the data size, e.g., the difference, it saves spaces. The initial state of the trace, which is stored at the head of the
buffer, will be destroyed if the trace is wrapped around, and thus making the rest of the trace not decompressable.

1) Periodical Triggering Concept: To solve the problem, we adopt a periodical triggering technique. The concept is to divide the entire trace into several independent small traces, as shown in Fig. 5. Since every trace has its own initial state, destroying the initial state of one trace does not affect other traces. This technique can be accomplished by periodically triggering a new trace. Therefore, it can be easily accomplished by the existing trace compression engines with minor modification to their control circuitry.

To support periodical triggering, the circular buffer is partitioned into segments, with each segment storing one small trace. The damage due to wrapping around is thus limited to one segment. Therefore, if the total number of the segments is N, after the first wrapping-around occurs (the buffer becomes full), there will be N−1 segments containing valid traces, and there will be one segment where the old trace is overwritten by the new trace. Assuming the trace size is random, half of this segment will contain valid new trace on the average. Therefore, the averaged total buffer utilization is \((2N−1)/2N\) (i.e., \((N−1)/N + 1/2N\)). As a result, the more the segments, the better the buffer utilization ratio. On the other hand, the more segments, the less the effective trace depth since there are more initial data, which are not compressed, populating the buffer. Therefore, tradeoff has to be made among the segment number, the buffer utilization ratio and the effective trace depth. We will perform such tradeoff analysis in Section V-B.

2) Circular Buffer Management: To decompress those traces (segments) in a circular buffer, we must know where the traces are in the circular buffer and which one is the oldest trace. This task is not straightforward because the trace lengths are variable due to the nature of compression. Therefore, a header position table is used to keep track the location of each trace, as shown in Fig. 6. This table consists of sixteen header position registers, which allows us to support up to 16 segments in the buffer. In addition, there is an oldest header register to point to the oldest trace. This helps the decomposition software to identify the location of the oldest trace so that it can decompress the trace in time order. For example, in Fig. 6(a), when the circular buffer is not full, this register points to the 1st trace. After, in Fig. 6(b), the buffer is full and trace 17th wraps around the circular buffer, the first trace is damaged because the initial state is overwritten. Then, the oldest header register is adjusted to point to the second trace. If necessary, more header position registers can be allocated to support more segments in larger buffer.

IV. BUS TRACER ARCHITECTURE

This section presents the architecture of our bus tracer. We first provide an overview of the architecture for the post-T trace. We then discuss the three major compression methods in this architecture. Finally, we show the extension of the post-T architecture to support the pre-T trace.

A. Post-T Tracer Architecture Overview

Fig. 8 is the bus tracer overview. It mainly contains four parts: Event Generation Module, Abstraction Module, Compression Modules, and Packing Module. The Event Generation Module controls the start/stop time, the trace mode, and the trace depth of traces. This information is sent to the following modules. Based on the trace mode, the Abstraction Module abstracts the signals in both timing dimension and signal dimension. The abstracted data are further compressed by the Compression Module to reduce the data size. Finally, the compressed results are packed with proper headers and written to the trace memory by the Packing Module.

1) Event Generation Module: The Event Generation Module decides the starting and stopping of a trace and its trace mode. The module has configurable event registers which specify the triggering events on the bus and a corresponding matching circuit to compare the bus activity with the events specified in the
event registers. Optionally, this module can also accept events from external modules. For example, we can connect an AHB bus protocol checker (HPChecker) [12] to the Event Generation Module, as shown in Fig. 8, to capture the bus protocol related trace.

Fig. 7 is the format of an event register. It contains four parameters: the trigger conditions, the trace mode, the trace direction, and the trace depth. The trigger conditions can be any combination of the address value, the data value, and the control signal values. Each of the value has a mask field for enabling partial match. For each trigger condition, designers can assign a desired trace mode, e.g., Mode FC, Mode FT, etc., which allows the trace mode to be dynamically switched between events. The trace direction determines the pre-T/post-T trace. The trace depth field specifies the length of trace to be captured.

2) Abstraction Module: The Abstraction Module monitors the AMBA bus and selects/filters signals based on the abstraction mode. The bus signals are classified into four groups as mentioned in Section III-A1. Then, depending on the abstraction mode, some signals are ignored, and some signals are reduced to states. Finally, the results are forwarded to the Compression Module for compression.

3) Compression Module: The purpose of the Compression Module is to reduce the trace size. It accepts the signals from the abstraction module. To achieve real-time compression, the Compression Module is pipelined to increase the performance. Every signal type has an appropriate compression method, as shown in Table III. The program address is compressed by a combination of the branch/target filtering, the dictionary-based compression, and the slicing. The data address and the data value are compressed by a combination of the differential and encoding methods. The ACS and PCS signals are compressed by the dictionary-based compression. Details will be discussed in Section IV-B.

4) Packing Module: The Packing Module is the last phase. It receives the compressed data from the compression module, processes them, and writes them to the trace memory. It is responsible for three jobs: packet management, circular buffer management, and mode change control. For packet management, since the compressed data length and type are variable, every compressed data needs a header for interpretation. Therefore, this step generates a proper header and attaches it to each compressed datum. In this paper, we call a compressed data with a header as a packet. Since the header generation takes time, to avoid long cycle time, the header generation is implemented in one pipeline stage. For circular buffer management, it manages the accesses to the trace memory. Since the size of a packet is variable but the data width of the trace memory is fixed, this module collects the trace data in a first-input, first-output (FIFO) buffer and outputs them to the trace memory until the data size in the FIFO buffer is equal/larger than the data width. If the tracing stops and the data size in the FIFO buffer is smaller than the data width, one additional cycle is required to output the remaining data to the trace memory. When the tracer is in the pre-T trace mode, this module also maintains the header position table mentioned in Section III-B-II. For mode change control, it manages the insertion of the special packet (called mode-change packet) that distinguishes the current mode from the previous mode. Details are discussed as follows.

Dynamic mode change can be achieved by changing the mode in the abstraction module. Designers can achieve this by setting the desired trace mode in the event register. However, since the header of each packet does not include the mode information because of space reduction, the decompression software cannot tell how to decompress the packets. Therefore, there must be a mode-change packet, that indicates the trace mode, placing between two tracers belonging to two different modes. The format
of this packet is shown in Fig. 9. This packet contains the mode of the next trace and the current bus BSM state. Since mode change might occur in the middle of a BSM transition, it is necessary for the decompression software to know the start BSM state of the next mode.

It is very important to insert this packet at the right time. Since the tracer is divided into several pipeline stages, during mode change, there are two trace data belonging to two modes in the pipeline stages. The insertion of the mode-change packet must wait until the trace data belonging to the previous mode to be processed. It is achieved by the mode change controller in the Packing Module. It accepts the mode change signal from the Event Generation Module and monitors the Abstraction Module and the Compression Module. When the last-cycle datum of the previous mode is processed and created as a packet, the mode-change packet is inserted into the FIFO with that packet at the same cycle. The reason of writing the two packets at the same time is to avoid pipeline stall due to inserting the mode-change packet, since the pipeline stall will prevent the bus tracer from accepting new input data and cause discontinuous traces.

B. Compression Mechanism

Although the Abstraction Module can reduce the trace size, the remaining trace volume is still very large. To reduce the size, the data compression approaches are necessary. Since the signal characteristics of the address value, the data value, and the control signals are quite different, we propose different compression approaches for them.

1) Program Address Compression: We divide the program address compression into three phases for the spatial locality and the temporal locality. Fig. 10 shows the compression flow. There are three approaches: branch/target filter, dictionary-based compression, and slicing.

2) Branch/Target Filtering: This technique aims at the spatial locality of the program address. Spatial locality exists since the program addresses are sequential mostly. Software programs (in assembly level) are composed by a number of basic blocks and the instructions in each basic block are sequential. Because of these characteristics, Branch/target filtering can records only the first instruction’s address (Target) and the last instruction’s address (Branch) of a basic block. The rest of the instructions are filtered since they are sequential and predictable.

3) Dictionary-Based Compression: To further reduce the size, we take the advantage of the temporal locality. Temporal locality exists since the basic blocks repeat frequently (loop structure), which implies the branch and target addresses after Phase 1 repeat frequently. Therefore, we can use the dictionary-based compression. The idea is to map the data to a table keeping frequently appeared data, and record the table index instead of the data to reduce size. Fig. 11 shows the hardware architecture. The dictionary keeps the frequently appeared branch/target addresses. To keep the hardware cost reasonable, the proposed dictionary is implemented with a CAM-based FIFO. When it is full, the new address will replace the address at the first entry of FIFO. For each input datum (din), the comparator compares the datum with the data in the dictionary (table[i]). If the datum is not in the table (match = Miss), the datum (uncompressed_data) is written into the table and also recorded in a trace. Otherwise (match = Hit), the index (match_index) of the hit table entry is recorded instead of the datum.

The hit index can be further compressed. As we know, a basic block is composed by a target address and a branch address, and the branch instruction address appears right after target instruction address. By the fact that basic blocks repeat frequently, if the target address is hit at the table entry i, the branch address will hit at the table entry (i + 1), since these entries are stored in the dictionary in a FIFO way. Therefore, instead of recording the hit index of that branch address, we create a special header, called the continuous hit, to represent that branch address if it meets this condition. This is the packet format 1 in Fig. 10.

4) Slicing: The miss address can also be compressed with the Slicing approach. Because of the spatial locality, the basic blocks are often near each other, which means the high-order bits of branch/target addresses nearly have no change. Therefore, the concept of the Slicing is to reduce the data size by recording only the different digits of two
consecutive miss addresses. To implement this concept in hardware, the address is partitioned into several slices of an equal size. The comparison between two consecutive miss addresses is at the slice level. For example, there are three address sequences: A(0001_0010_0000), B(0001_0010_0110), C(0001_0110_0110). At first, we record instruction A’s full address. Next, since the upper two slices of address B are the same as that of the address A, only the least-significant slice is recorded. For address C, since the most significant slice is the same to that of the address B, only the lower two slices are recorded.

Fig. 12 shows the hardware architecture. It has the register REG storing the previous data (\(d_{i-1}\)). The slice comparator compares the slices of the current datum (\(d_i\)) and the previous datum and produces the identical slice number (\(size_i\)). This information is forwarded to the packing module to generate the proper header. This is the packet format 3 in Fig. 10.

Table IV shows an example of the compression approaches in Phases 2 and 3. At time 1, since the address (0x00008020) cannot be found in the dictionary, it is inserted into the dictionary entry 0 and is recorded in a trace. At time 2, the address (0x00008030) is also a miss address and inserted into dictionary entry 1. However, after slicing, since only the lower two slices are different, only the address 0x30 is recorded. At time 5, because the address (0x00008020) has been stored in the dictionary entry 0 at time 1, only the index 0x0000 is recorded. At time 6, since the address (0x00008030) also has been stored in the dictionary entry 1, and its index is the previous address plus 1, we do not have to record anything except the header (as the packet format 1 in Fig. 10), which indicates this is a hit address and this meets the continue index condition.

5) Data Address/Value Compression: Data address and data value tend to be irregular and random. Therefore, there is no effective compression approach for data address/value. Considering using minimal hardware resources to achieve a good compression ratio, we use a differential approach based on the subtraction.

Fig. 13 shows the hardware compressor. The register REG saves the current datum \(d_i\) and outputs the previous datum \(d_{i-1}\). By comparing the current datum with the previous data value, the three modules \(comp, differential\), and \(sizeof\) output the encoded results. The \(comp\) module computes the sign bit (\(sign_{diff}\)) of the difference value. The differential module calculates the absolute difference value (\(value\)). Since the absolute difference between two data value may be small, we can neglect the leading zeros and use fewer digits to record it. Therefore, the \(sizeof\) module calculates the nonzero digit number (\(sizei\)) of the difference. Finally, the encoded datum is sent to the packing module along with \(sizei\).

For simple hardware implementation, the digit number of an absolute difference is limited to four types, as Fig. 14 shows. The header indicates the data trace format. If the difference is larger than 65535 (\(2^{16} - 1\)), the bus tracer record the uncompressed full 32-bit data value. Otherwise, the bus tracer uses 4-, 8-, or 16-bit length to record the absolute differences, whichever is appropriate.

6) Control Signal Compression: We classify the AHB control signals into two groups: access control signals (ACS) and protocol control signals (PCS). ACS are signals about the data access aspect, such as read/write, transfer size, and burst operations. PCS are signals controlling the transfer behavior, such as master request, transfer type, arbitration, and transfer response.

Control signals have two characteristics. First, the same combinations of the control signals repeat frequently, while other
combinations happen rarely or never happen. The reason is that many combinations do not make sense in a SoC. It depends on the processor architecture, the cache architecture, and the memory type. Therefore, the IPs in a SoC tend to have only a few types of transfer despite the bus protocol allows for many transfer behaviors. Second, control signals change infrequently in a transaction.

Because of these two characteristics, ACS/PCS are suitable for dictionary-based compression. The idea is to treat the signals in ACS/PCS as one group. Since the variations of transfer types are not much and transfer types repeat frequently, we can map them to the dictionary with frequently transfer types to reduce size. For example, the original size of ACS is 15 bits. If we use 3-bit to encode the signal combinations of ACS, we can reduce size. For example, the original size of ACS is 15 bits. If we use 3-bit to encode the signal combinations of ACS, we can reduce size by $(1 - 3/15) \times 100\% = 80\%$. To simplify the hardware design for cost consideration, this dictionary is also implemented as a FIFO buffer. With this approach, the dictionary adapts itself when the ACS/PCS behaviors change.

Please notice that, in full-signal level trace, both ACS and PCS are compressed by the dictionary-based compression without abstraction. In bus-state level trace, the PCS are first abstracted into states by the BSM model and then compressed by the dictionary-based compression.

C. Extension of the Post-T Trace Architecture to Support the Pre-T Trace

The bus tracer described in Sections IV-A and IV-B is for the post-T trace. We now extend the bus tracer to support the pre-T trace with the technique of periodical triggering.

The concept of periodical triggering is to break the relationship between the new trace and the previous trace. This can be achieved by resetting the internal data structure for data encoding. For example, the register REG keeping the previous data in the slicing (see Fig. 12) and the differential compressor (see Fig. 13) must be reset. Also, the table in the dictionary-based compression (see Fig. 11) is the same.

We use Fig. 15 to illustrate the concept. It is an example showing the periodical triggering of the differential compression. The encoded result (the difference $\Delta f_{i}$) is produced by subtracting the previous data ($D_{i-1}$) from the current data ($D_{i}$). For example, the encoded data is $2 = 5 - 3$ at time $T$. If the trace mode does not change, the current data is registered for encoding the new data at the next cycle. Otherwise, the flush signal asserts. Then, the register keeping the previous data is reset and a new trace begins. For example, at time $T + 2$, $D_{i}$ (11) is recorded in the uncompressed format by subtracting 0 from it, which serves as an initial state for the new trace.

Please notice that no data is lost during the reset, though the data storage cannot accept new input data while it is reset. For example, the register in Fig. 15 takes 1 cycle ($T + 2$) to reset, during that cycle, the input data ($D_{i}$ with value 11) is recorded in uncompressed format.

To implement the periodical triggering concept, we add an extension hardware to the original tracer, as shown in Fig. 16. The Triggering Module decides the time to start a new trace based on the trace length (a segment size). It asserts the flush signal when a new trace begins. Since the compression module is divided into several pipeline stages, the flush signal is also pipelined to reset each pipeline stage in order. This is necessary since the Compression Module requires several cycles to process the data belonging to the previous trace.

D. Integration Into SoC

To integrate the bus tracer (including the trace memory) into a SoC, we can simply tap the bus tracer to the AHB bus, as shown in Fig. 17. The bus tracer can be controlled with an on-chip processor (option 1) or an external debugging host (option 2). For option 1, the processor configures the bus tracer via a bus slave interface. After the bus tracer compresses and stores the

![Fig. 15. Example of periodical triggering in differential compression.](image)

![Fig. 16. Extension architecture for supporting post-T trace. Below is the extension to the original bus tracer.](image)

### Table V

<table>
<thead>
<tr>
<th>Specification of the Implemented SYS-HMRBT Bus Tracer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
</tr>
<tr>
<td>Trace Mode</td>
</tr>
<tr>
<td>Trace Direction</td>
</tr>
<tr>
<td>Input AHB signals</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Output trace width</td>
</tr>
<tr>
<td>Pipeline stage</td>
</tr>
<tr>
<td>Max. # of masters</td>
</tr>
<tr>
<td>FIFO buffer</td>
</tr>
</tbody>
</table>

* The event generation module contains two event registers. Each uses about 1,500 gates.

### Table VI

<table>
<thead>
<tr>
<th>Syntheses Results Under TSMC 0.13-μm Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Bus interface unit</td>
</tr>
<tr>
<td>Event Gen. Module*</td>
</tr>
<tr>
<td>Abstraction Module</td>
</tr>
<tr>
<td>Compression Module</td>
</tr>
<tr>
<td>Packing Module</td>
</tr>
<tr>
<td>Periodical triggering</td>
</tr>
<tr>
<td>FIFO buffer (512 bits)</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

* The event generation module contains two event registers. Each uses about 1,500 gates.
TABLE VII
TRACE COMPRESSION RATIO AT DIFFERENT TRACE MODES

<table>
<thead>
<tr>
<th>Program</th>
<th>Uncompressed trace size</th>
<th>Mode FC</th>
<th>Mode FT</th>
<th>Mode BC</th>
<th>Mode BT</th>
<th>Mode MT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perpetual Calendar</td>
<td>910,000</td>
<td>77.58%</td>
<td>83.00%</td>
<td>90.05%</td>
<td>94.67%</td>
<td>96.99%</td>
</tr>
<tr>
<td>Fibonacci Sequence</td>
<td>910,000</td>
<td>78.32%</td>
<td>80.05%</td>
<td>87.91%</td>
<td>90.45%</td>
<td>94.73%</td>
</tr>
<tr>
<td>G.C.D.</td>
<td>910,000</td>
<td>83.22%</td>
<td>83.45%</td>
<td>89.14%</td>
<td>92.76%</td>
<td>95.68%</td>
</tr>
<tr>
<td>Towers of Hanoi</td>
<td>910,000</td>
<td>78.64%</td>
<td>80.61%</td>
<td>85.99%</td>
<td>88.58%</td>
<td>92.56%</td>
</tr>
<tr>
<td>Knight Problem</td>
<td>910,000</td>
<td>78.85%</td>
<td>80.51%</td>
<td>96.39%</td>
<td>97.16%</td>
<td>98.32%</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>910,000</td>
<td>78.27%</td>
<td>79.99%</td>
<td>90.64%</td>
<td>99.91%</td>
<td>99.80%</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>79.00%</td>
<td>81.26%</td>
<td>91.39%</td>
<td>93.81%</td>
<td>96.32%</td>
<td></td>
</tr>
</tbody>
</table>

* Trace 10,000 cycles for each benchmark.
* Uncompressed trace size = (91 x 10,000) = 910,000 bits
* Although there are one read data bus and one write data bus in AHB, only one of them could be active at any time. The bus tracer only traces the active one.

Fig. 17. Example of integrating the bus tracer into a SoC.

In order to achieve real-time tracing, the bus tracer is pipelined to meet the on-chip bus frequency. Since the trace data processing is stream-based, the bus tracer can be easily divided into more pipeline stages to meet aggressive performance requirements.

V. EXPERIMENTAL RESULTS

The specification of the implemented SYS-HMRBT bus tracer is shown in Table V. It has been implemented at C, RTL, FPGA, and chip levels. The synthesis result with TSMC 0.13-μm technology is shown in Table VI. The bus tracer costs only about 41 K gates, which is relatively small in a typical SoC. The largest component is the FIFO buffer in the packing module. The second one is the compression module. The cost to support both the pre-T and post-T capabilities (periodical triggering module) is only 1032 gates. The major component of the event generation module is the event register, which is roughly 1500 gates per register. The implementation in this paper has two event registers. More registers can be added if necessary. Compared with our previous work [13], the gate count is reduced by 31%. The reason is that this paper optimizes the ping-pong architecture by sharing most of the data path instead of duplicating all the hardware components. As for the circuit speed, the bus tracer is capable of running at 500 MHz, which is sufficient for most SoC’s with a synthesis approach under 0.13-μm technology. If a faster clock speed is necessary, our bus tracer could be easily partitioned into more pipeline stages due to its streamlined compression/packing processing flow.

In the rest of this section, we present the analysis of various system metrics of the bus tracer, such as trace resolution, compression quality, depth, trace memory size, and I/O pin count, etc. In addition, we provide a real example of embedding our bus trace into a 3-D graphics SoC chip for consumer electronics.

A. Analysis of the Trace and Hardware Characteristics

To evaluate the effectiveness of our bus tracer, we integrated it with ARM’s EASY (Example AMBA SYstem) SoC platform [14]. Five C benchmark programs were executed on this platform. The first 10,000 cycles of AHB signals (a mixture of setup and loop operations) were captured as a post-T trace under FC, FT, BC, BT, and MT trace modes, respectively. The results were shown in Table VII. The average compression ratios of these benchmark programs range from 79% for the most detailed mode FC (full signals, cycle-level) to 96% for the most abstract mode MT (master state, transaction-level). In between are the other modes with intermediate levels of abstraction.

The high compression ratio achieved by our bus tracer makes it possible to output the trace data to the outside world in real time via output pins. Table VIII shows the required minimal pin count for each trace mode, ranging from 7 to 21. Please note that the pin counts can be shared among different modes. For example, if there is 21 output pins available for the bus tracer, all five modes could be output in real time, whereas three modes (BC, BT, and MT) could be output in real time when there are
13 pins available. However, with only seven pins, the bus tracer could still output the trace at mode MT in real time. Therefore, our bus tracer allows designers to customize the pin resource and trace resolution for real time trace dumping to match a diverse range of debugging needs.

If output pins are not available, we can also store the trace data in an on-chip trace memory. Table IX analyzes the trace depth (the number of bus cycles) that a trace memory can accommodate under the five trace modes and four memory sizes (2/4/8/16 kB). The "full signal, no compression" column in the table, used as the comparison base, refers to storing the full bus signals directly in the memory without compression. This column is equivalent to the result of LEON3 AHBTRACE [5]. The mode FC of our tracer captures the same full signals with compression support. For a 2 kB trace memory, AHBTRACE captures 180 cycles while our tracer captures 821 cycles, which is 4.7× improvement. Our tracer captures even more cycles with intermediate-level abstract modes and up to 2453 cycles (13.6×) with the most abstract mode MT. With 16 KB trace memory, our tracer could capture 6619 and 18 657 cycles for modes FC and MT, respectively. Therefore, by properly configuring the event registers and selecting the appropriate trace mode, our tracer is able to capture a sufficient range of trace depths that are sufficient for typical cycle-level, bus-state and master-state debugging/monitoring activities with a small or moderate size of trace memory. Therefore, our bus tracer is capable of maximizing the trace memory capacity while allowing the designers to trade off between trace depth and resolution.

We further explored the dynamic mode change (DMC) feature of our bus tracer. Table X estimates the number of cycles that can be captured in each trace mode under four configurations of dynamic mode change in a 2 kB trace memory (base on the information in Table IX). For each configuration, the numbers in parenthesis show the size percentage of the five modes in the trace memory respectively. For example, configuration I captures trace segments under modes FC, FT, BC, BT, and MT, with each mode taking up 10%, 15%, 20%, 25%, and 30% of the trace memory respectively. The resulted depth of each segment is 82, 144, 261, 398, and 736 cycles, respectively. This experiment demonstrates that our bus tracer allows users to dynamically zoom in/out the observation of the bus for different level of details and for different periods of time, and thus it is capable of supporting versatile SoC development/debugging needs, such as module development, chip integration, hardware/software integration and debugging, system behavior monitoring, system performance/power analysis and optimization, etc.

Table XI compares the features of our tracer with related AHB bus tracers: ARM’s HTM [3], FS2 AMBA Navigator [4] and LEON3 AHBTRACE [5]. (Since the TC1775 trace module in TriCore [6] and ARM ETM [7], reviewed in Section II, are for processor tracing instead of bus tracing, we do not compare our work with them in the experiment.) Of the three related works, ARM’s HTM is the only one which attends to compress traces. It uses the slicing technique to reduce the trace size of the program address (Paddr) and control signals (Ctrl). On the other hand, the data address (Daddr) trace size is reduced by removing the higher order zeros. However, the data value (Dvalue) trace is not supported by HTM. Compared with HTM, our tracer supports all four kinds of signals (Paddr, Daddr, Dvalue, and Ctrl), all with more aggressive compression algorithms. On the other hand, AMBA Navigator and AHBTRACE support all four kinds of signals, but do not provide any compression support.

For the trace direction, AMBA Navigator and AHBTRACE support only the post-T traces, while HTM and our tracer support both pre-T and post-T traces. As for the multi-resolution support, HTM and AMBA Navigator have limited abstraction capability in the timing dimension. They filter signals when the bus state is in the IDLE or BUSY cycles. On the other hand, our tracer supports abstraction in both timing and signal dimensions, which provides more versatile debugging/monitoring functionalities and better trace compression ratio. In addition, our tracer allows dynamic mode change, which is a unique feature among all AHB bus tracers.

It is not an easy task to conduct quantitative comparison. HTM and AMBA Navigator are commercial products which we do not have access to, and there is no information about their trace quality available in literature. However, based on the qualitative comparison in Table XI, we could reasonably conclude that our tracer supports both more features and better compression quality. On the other hand, the technical information

### Table IX

<table>
<thead>
<tr>
<th>Trace memory size</th>
<th>Full signals, no compression (AHBTRACE [5])</th>
<th>Mode FC</th>
<th>Mode FT</th>
<th>Mode BC</th>
<th>Mode BT</th>
<th>Mode MT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 KB</td>
<td>180</td>
<td>821 (4.7x)</td>
<td>961 (5.3x)</td>
<td>1,303 (7.2x)</td>
<td>1,591 (8.8x)</td>
<td>2,453 (13.6x)</td>
</tr>
<tr>
<td>4 KB</td>
<td>360</td>
<td>1,661 (6.6x)</td>
<td>1,925 (5.3x)</td>
<td>2,641 (7.3x)</td>
<td>3,243 (9.0x)</td>
<td>5,021 (13.9x)</td>
</tr>
<tr>
<td>8 KB</td>
<td>720</td>
<td>3,303 (6.6x)</td>
<td>3,885 (5.4x)</td>
<td>5,333 (7.4x)</td>
<td>6,559 (9.1x)</td>
<td>10,147 (14.1x)</td>
</tr>
<tr>
<td>16 KB</td>
<td>1440</td>
<td>6,619 (6.6x)</td>
<td>7,761 (5.4x)</td>
<td>10,719 (7.4x)</td>
<td>13,001 (9.0x)</td>
<td>18,657 (13.0x)</td>
</tr>
</tbody>
</table>

The number in parentheses indicates the trace depth improvement over the noncompression method for the corresponding trace memory size.

### Table X

<table>
<thead>
<tr>
<th>Dynamic mode change configuration</th>
<th>Mode FC</th>
<th>Mode FT</th>
<th>Mode BC</th>
<th>Mode BT</th>
<th>Mode MT</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: (10%, 15%, 20%, 25%, 30%)</td>
<td>82</td>
<td>144</td>
<td>261</td>
<td>398</td>
<td>736</td>
<td>1,621</td>
</tr>
<tr>
<td>II: (15%, 20%, 25%, 30%, 10%)</td>
<td>123</td>
<td>192</td>
<td>326</td>
<td>477</td>
<td>724</td>
<td>1,364</td>
</tr>
<tr>
<td>III: (20%, 25%, 30%, 10%, 15%)</td>
<td>164</td>
<td>240</td>
<td>391</td>
<td>559</td>
<td>891</td>
<td>1,322</td>
</tr>
<tr>
<td>IV: (25%, 30%, 10%, 15%, 20%)</td>
<td>205</td>
<td>288</td>
<td>430</td>
<td>639</td>
<td>941</td>
<td>1,353</td>
</tr>
<tr>
<td>V: (30%, 10%, 15%, 20%, 25%)</td>
<td>246</td>
<td>286</td>
<td>409</td>
<td>558</td>
<td>851</td>
<td>1,469</td>
</tr>
</tbody>
</table>
of AHBTRACE is available for comparison. Since it does not provide any compression support, its trace size is equivalent to the “No compression” column of Table IX. Our tracer is capable of capturing 4.6 up to 14.1 times of the trace depth than AHBTRACE, depending on the selected trace mode and trace memory size.

### B. Analysis of Pre-T Trace Configuration and Quality

As discussed in Section III-B-1, the pre-T trace requires more care than the post-T trace. There are tradeoffs between the number of segments in the circular buffer, the buffer utilization, and the effective trace depth for the pre-T trace. Table XII lists the buffer utilization and the trace depth for various segment configurations. The size of the buffer is 1 kB. The first row is the case of AHBTRACE which serves as the base design for comparison. The buffer utilization of it is 100% since the data replacement wastes no space in wrapping around. The second through fifth rows are the cases of our periodical triggering in mode FC with 1, 2, 4, 8, and 16 segments, respectively. For each case, we show the trace buffer utilization, trace depth in cycles, and the relative trace depth with respect to the base design. The experiment shows that all our cases have relative trace depth higher than the base design, ranging from 2.32 to 3.98. Note that although the sixteen-segment case has the largest buffer utilization (96.9%) among all our cases, it does not yield the longest trace depth. The reason is that there are many (sixteen) small traces in the buffer, with each trace containing non-compressed initial state in its header and thus lowering the global trace compression quality. Instead, the eight-segment case, having a slightly lower buffer utilization (93.8%), achieves the longest relative trace depth of 3.98.

Finally, let’s compare the compression efficiency of pre-T and post-T traces. Since pre-T tracing uses the same compression mechanisms as post-T tracing except that the former uses the periodical triggering approach, which causes multiple initial uncompressed data in the trace memory, to solve the wrapping-around problem, we expect the compression ratio of the pre-T traces to be slightly smaller than the compression ratio of post-T traces. Table XIII shows the comparison for various trace memory sizes. The first row is the result for post-T traces, and the second row is the result of pre-T traces with the eight-segment configuration as suggested in the previous paragraph. Note that in order to have a fair comparison, we assume the full buffer utilization for both post-T traces and pre-T traces. For the 1 kB trace memory, the compression ratio of pre-T traces is about 6.71% inferior to that of post-T traces. However, the difference is reduced as the trace memory size increases: only 3.75% for 16 kB trace memory. Practically speaking, the difference (from 3.75% to 6.71%) between the pre-T and post-T traces is not significant in most debugging/monitoring needs. Should such difference matters, the designer should choose a larger trace memory as permitted by the global cost budget to minimize such difference.

### C. Case Study: 3-D Graphics SoC Integration

We have integrated our bus tracer with a 3-D graphics (3DG) SoC [15] at both RTL, FPGA, and chip levels. Fig. 18(a) shows a demonstration on ARM’s Versatile FPGA prototyping platform [14]: a 3DG benchmark being rendered by the SoC to an LCD monitor. The related hardware modules are shown in Fig. 18(b). The hardware blocks are divided into two parts: a mother board and an FPGA board. On the mother board are a microprocessor ARM926EJ with peripherals, which executes Linux operating
YANG et al.: ON-CHIP AHB BUS TRACER WITH REAL-TIME COMPRESSION 583

Fig. 18. Chip and prototyping of the 3-D graphics SoC. (a) The ARM’s Versatile integrated development board. (b) The SoC block diagram in the Versatile. (c) The graphic user interface of the trace analyzer. c1 shows the decompressed waveform. c2 shows the address/data distribution. c3 shows the protocol analysis in pie chart. c4 shows the master behavior analysis. (d) Chip photo of the 3-D graphics SoC with our bus tracer and the synthesis results.

The 3DG SoC has been fabricated with TSMC 0.18-μm technology. Fig. 18(d) shows the chip synthesis results and the chip photo. The bus tracer is the C-shaped block located near the center right edge and the trace memory is located at the bottom right edge. The entire SoC has about 986 K gates while the bus tracer costs only about 54 K gates, which is only 5.5% of the SoC. The SoC runs at 139 MHz while the bus tracer itself is capable of running at more than 400 MHz. These results indicate that our bus tracer does not impair the system performance when it is connected to the on-chip bus, and also its overhead in a SoC is very minor.

The bus tracer and the protocol checker successfully identified several bugs related to the bus wrappers of GE and RE during the FPGA verification. In the beginning, the bus tracer was configured to capture the pre-T trace continuously and save it in the trace memory circularly since there was no prior knowledge about when a bug would occur. The pre-T tracing stopped after the protocol checker triggered a protocol violation event. The pre-T trace was then read out for analysis of the cause. After the bug was identified and fixed, the bus tracer was configured to capture the post-T trace since the location of the bug was known at this moment. The same benchmark was executed again. The post-T trace was then read out to confirm if the problem had been resolved.

The bus tracer not only helped the designer to debug the SoC during design integration, but also helped the user of the SoC to monitor the real time system behaviors when the SoC is working, either in FPGA or chip. Fig. 18(c) shows several screen shots of our trace analyzer, which analyzes the bus utilization ratios of on-chip bus components, the address/data distribution, the primary transfer type, etc., for a portion of the 3DG benchmark’s execution. The user could use these information for performance analysis and tuning.

VI. CONCLUSION

We have presented an on-chip bus tracer SYS-HMRBT for the development, integration, debugging, monitoring, and tuning of AHB-based SoC’s. It is attached to the on-chip AHB bus and is capable of capturing and compressing in real time the bus traces with five modes of resolution. These modes could be dynamically switched while tracing. The bus tracer also supports both directions of traces: pre-T trace (trace before the triggering event) and post-T trace (trace after the triggering event). In addition, a graphical user interface, running on a host PC, has been developed to configure the bus tracer and analyze the captured traces. With the aforementioned features, SYS-HMRBT supports a diverse range of design/bugging/monitoring activities, including module development, chip integration, hardware/software integration and debugging, system behavior monitoring, system performance/power analysis and optimization, etc. The users are allowed to tradeoff between trace granularity and trace depth in order to make the most use of the on-chip trace memory or I/O pins.

SYS-HMRBT costs only 42 K gates, making it an valuable and economical investment in a typical SoC. It runs at 500 MHz in TSMC 0.13-μm technology, which satisfies the requirement of real-time tracing. Experiment results show it achieves high compression ratio ranging from 79% to 96% depending on the trace mode. The bus tracer has been successfully integrated into a 3-D graphics SoC for digital TV and verified in both FPGA and test chip levels.
In the future, we would extend this work to more advanced buses/connects such as AXI or OCP. In addition, with its real time abstraction capability, we would like to explore the possibility of bridging our bus tracer with ESL design methodology for advanced hardware/software codevelopment/debugging/monitoring/analysis, etc.

REFERENCES


Fu-Ching Yang (S’06–M’09) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from the National Sun Yat-Sen University, Kaohsiung, Taiwan, in 2003, 2004, and 2009, respectively.

He is currently a Senior Engineer with the Global Unichip Corporation. He was with the Department of Computer Science and Engineering, National Sun Yat-Sen University. His research interests include SoC debugging methodology, trace compression algorithm development, and microprocessor design/verification.

Yi-Ting Lin received the B.S. degree in computer science and information engineering from Nation Chiao Tung University, Hsinchu, Taiwan, in 2002, and the M.S. degree in computer science and engineering from the National Sun Yat-Sen University, Kaohsiung, Taiwan, in 2004.

He is currently a Senior Engineer with the Institute for Information Industry, Taiwan. He was with the Department of Computer Science and Engineering, National Sun Yat-Sen University. His research interests include SoC infrastructure, microprocessor design, and 3-D graphics system.

Chung-Fu Kao received the B.S. degree in computer science and information engineering from Tamkang University, Taiwan, in 1998, and the M.S. and Ph.D. degrees in computer science and engineering from National Sun Yat-Sen University, Kaohsiung, Taiwan, in 2000 and 2007, respectively.

He is currently a Senior Software Engineer with the Marvell. He was with the Department of Computer Science and Engineering, National Sun Yat-Sen University. His research interests include microprocessor/SoC debugging, SoC platform design, and hardware/software co-verification.

Ing-Jer Huang (S’88-M’95) received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1986, and the M.S. and Ph.D. degrees in computer engineering from the University of Southern California, Los Angeles, in 1989 and 1994, respectively.

He is currently a Professor with the Department of Computer Science and Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan. His research interests include computer architectures, system-on-a-chip designs, design automation, system software, embedded systems, and hardware/software co-design/verification. He has been actively involved in academic, educational and industrial activities. He has extensive collaboration with several IC-related industries. Dr. Huang is a member of the Association for Computing Machinery (ACM).