Real-Time Address Trace Compression for Emulated and Real System-on-Chip Processor Core Debugging

Bojan Mihajlović
McGill University
Dept. of Electrical and Computer Engineering
Montreal, Quebec, Canada
bojan.mihajlovic@mcgill.ca

Željko Žilić
McGill University
Dept. of Electrical and Computer Engineering
Montreal, Quebec, Canada
zeljko.zilic@mcgill.ca

ABSTRACT

In the multicore era, capturing execution traces of processors is indispensable to debugging complex software. The inability to transfer vast amounts of trace data off-chip without significant slow-down has impeded the debugging of such software, in both pre-silicon emulation and in real designs. We consider on-chip trace compression performed in hardware to reduce data volume, using techniques that exploit inherent higher-order redundancy in address trace data. While hardware trace compression is often restricted to poor or moderate performance due to area and memory constraints, we present a parameterizable scheme that leverages the resources already found on existing platforms. Harnessing resources such as existing trace buffers on CPUs, and unused embedded memory on FPGA emulation platforms, our trace compression scheme requires only a small additional hardware area to achieve superior compression ratios.

Categories and Subject Descriptors: E.4 [Coding and Information Theory]: Data compaction and compression

General Terms: Algorithms, Verification

Keywords: Emulation, Software debugging

1. INTRODUCTION

Modern systems-on-chip (SoCs) have evolved significantly in recent years. Multiple processors (MPSoCs), memory controllers, and graphics processing units (GPUs) are increasingly found on-chip, often running in different clock domains to optimize their performance and power profiles. Fully utilizing the available resources of MPSoCs requires that programmers create multi-threaded, often interdependent software suited to multiprocessors. Debugging such software can potentially be impossible using traditional debugging methods, which were created around a single processor core in mind. It has already been recognized that system-centric, rather than processor-centric solutions are needed to address future software debug challenges [10]. Since traditional methods violate timing requirements in multi-threaded software, non-invasive methods of debugging software are being studied as replacements. A dedicated debug infrastructure, also referred to as *on-chip emulation* [13] should be used to allow modern SoC software to execute at-speed while transparently logging debug events.

On-chip emulation places debug hardware inside the chip, aiming to transparently capture relevant data in real-time when user-configurable *triggers* are activated. Captured data is known as a *trace*, the most useful of which is an address trace (or execution trace) consisting of executed instruction addresses. It has long been established that examining such a trace can help determine behaviors that lead to software faults [8]. Such analysis is commonly performed off-line in a remote debugging scenario, depicted in Figure 1 for either real SoC or FPGA-emulated processor cores.

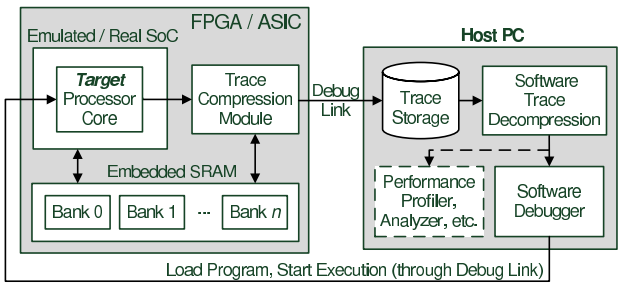

Figure 1: Trace-based Debugging Scenario

Due to its hardware cost, on-chip emulation has historically been used to debug software unsuited to other debug methods, namely hard real-time software [15]. However, it is now being considered as a solution for debugging the multi-threaded software increasingly seen on MPSoCs. Traditional debugging methods rely on execution control to achieve a necessary observability while software is executing. In such a scenario, software execution is stopped and registers are observed before execution is resumed. This is unsuitable for multiprocessors because multi-threaded software often contains interdependent threads, where halting a single thread would alter timing and potentially cause or mask faults.

One of the problems with address tracing is in the immense amount of data that must be either stored on-chip or transferred off-chip in real-time. Consider an example of a program counter (PC) trace of a 32-bit processor, capable of 1 clock per instruction (CPI), that is emulated on an FPGA at 100 MHz. The resulting 400 MB/s of data would quickly fill an on-chip trace buffer, or exceed the bandwidth of all but the most expensive communication links. When tracing

GLSVLSI'11, May 2–4, 2011, Lausanne, Switzerland.

anything but the smallest portions of code, a solution to this problem requires that the volume of trace data be reduced. That would allow the data to be transferred off-chip with links of modest bandwidth. When an adequate reduction of trace data is not possible, it typically means that the system must be slowed down to accommodate the speed at which traces can be extracted. This problem can be seen in [6], where the authors report that trace extraction lasted 34x longer than execution time for an FPGA-emulated processor. Likewise, work in [7] demonstrates that system emulation speed is reduced 10x with tracing enabled. The speed of trace extraction can then be seen as the bottleneck of multi-threaded software debugging (or verification). If trace extraction can be sped up, debugging time can be reduced.

This paper presents a method for compressing address traces in real-time that capitalizes on the inherent higher-order redundancy of trace streams. While other compression methods avoid using on-chip memory, our method uses memory that is already available as a compression dictionary. Many modern ASIC SoC designs incorporate a trace buffer memory into which traces can be captured and stored. If this memory can instead be used as a dictionary, and increased trace compression ratios realized, compressed trace data can be continuously streamed off-chip in real-time. While our design can be implemented on either ASICs or FPGAs, it presents some distinct advantages when used with an FPGA emulation design flow. Due to the increasing prominence of FPGA emulation in architectural exploration [7] and validation [6], our proposed scheme also capitalizes on the embedded SRAM memories found in modern FPGAs used for SoC prototyping. These memories can be underutilized when the emulated design does not require 100% of available FPGA resources, which is typically the case. We present a compression scheme that can scale to a range of dictionary sizes in order to exploit as much embedded SRAM as is available. The algorithm is shown to improve compression ratios without significant hardware cost, and enables real-time trace compression and extraction.

2. RELATED WORK

Address trace compression is a topic that has lately received significant attention. Traditionally, the most capable trace compression schemes, such as [5], work by compressing traces in software by utilizing techniques that are able to eliminate data redundancy at the cost of memory usage, algorithmic complexity, and consequently, processing time. Software trace compression techniques have the luxury of being able to implement a multitude of compression techniques and pick the smallest representation as the final output. Multi-pass algorithms and transforms are also possible when there are no time constraints, and the resulting data can be coded using a minimum-length code. Such software methods are, however, not suitable for compressing real-time traces during at-speed execution. Rather, they are adapted to compressing traces that have already been collected.

Real-time address trace compression is typically implemented in hardware and faces increased constraints over software methods. To guarantee sufficient performance, algorithms must be of relatively modest complexity and be capable of being pipelined or of having other performance enhancements applied. The lack of sizable on-chip memory also limits the types of algorithms that can be used.

Prior work in real-time address trace compression has ranged from relatively simple methods requiring little hardware area, to advanced multi-stage methods. Simple techniques widely employ differential address encoding that costs little hardware area, but suffers from poor performance and does not enable at-speed tracing. Advanced techniques offer better performance by utilizing the ubiquitous Lempel-Ziv (LZ) [18] coding variants, or even low-complexity transforms such as Move-to-Front (MTF), in addition to differential techniques. In [11], the authors use a 3-stage technique which filters non-sequential addresses, encodes remaining addresses as differences, then applies LZ compression. In [17], consecutive addresses are first reduced to a combination of a starting address and length, where least significant bits (LSBits) of the address are encoded as a difference, and a two-level MTF scheme is applied to both the length and the most significant bits (MSBits) of the address. Although geared toward embedded logic analyzers, encoders based upon both MTF and LZ methods are proposed in [1]. Current commercial tools include ARM Embedded Trace Macrocell (ETM) [2] and Infineon Multi-Core Debug Solution (MCDS) [12], both of which only support the use of trace buffers. While both tools employ differential address encoding, ETM also eliminates sequential addresses.

Due to memory limitations, most of the existing techniques have been unable to exploit higher-order data redundancy, caused by temporal and spatial locality over large periods of time and volume of space, respectively. Our technique differs in that it is able to garner experience by "remembering" more of the addresses and their sequences that pass through the compressor. Longer program executions should therefore benefit by exploiting the experience garnered over time to further reduce redundancy in encoding.

3. PROPOSED COMPRESSION SCHEME

The address trace compression scheme proposed here is designed with a particular software debugging scenario in mind. As seen in Figure 1, an emulated or real SoC containing a processor core (known as a *target*) together with a trace compression module, is connected to a *host* machine. The host is responsible for controlling software execution on the target and collecting its execution traces. Both the SoC and the compression module use some part of the embedded SRAM available on-board the FPGA or ASIC. Using a software debugger on the host, a tracing session is initiated by loading a program onto the target and starting its execution. Traces are collected in real-time, after which they are decompressed off-line by host software, allowing it to see the execution path taken by the target. To achieve this scenario, the trace data needs to be small enough to transfer from target to host in real-time through the debug link. It also needs to be small enough for the host to store in real-time onto a storage medium. While the speed of software decompression is not as important a consideration, the decompression of our scheme is conducive to a multi-threaded software implementation on the host. After address traces are decompressed, they can be sent back to the software debugger for inspection, and optionally fed into an automated performance analysis or profiling tool.

While software trace compression methods are computationally intense and use large amounts of memory, hardware-based trace compression must meet several constraints. Trace compression must occur in real-time if data is to be immediately transferred off-chip, must not require more than the

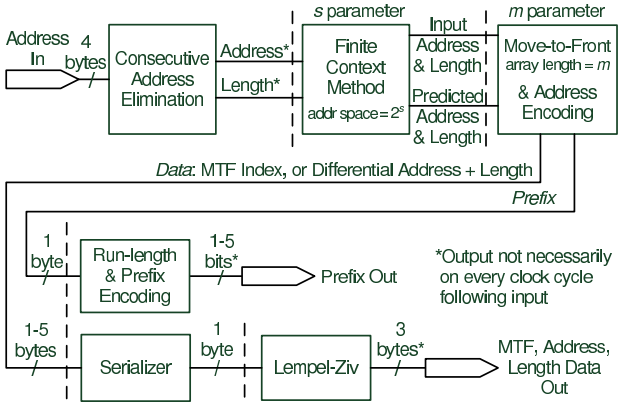

Figure 2: Multi-Stage Compression Scheme

small amounts of available on-chip memory, and its algorithm must be well-suited to hardware implementation in order to meet hardware area constraints. Our trace compression scheme, Figure 2, is designed to meet the above constraints for the described debug scenarios. While our scheme could be adapted to any processor core, we specifically target the characteristics of ubiquitous fixed instruction-length (RISC) processors to attain additional compression.

For debug scenarios involving both real and emulated cores, we investigate a parameterizable scheme suited to differing availabilities of hardware resources. In Figure 2, the variation between versions is represented by the 2-tuple (s, m), where 2^s is the size of the Finite Context Method address space, and m is the number of elements in the Move-to-Front array. Parameterization allows several options for harnessing leftover hardware area and memory on emulated SoCs, while ASIC designs can strike the appropriate area, speed, and cost trade-off. Scheme versions *v0* to *v4* considered in the remainder of the article correspond to $(s, m) = (16, 256)$, $(16, 128)$, $(14, 128)$, $(14, 64)$, and $(12, 64)$, respectively.

3.1 Consecutive Address Elimination

Since processors execute instructions consecutively until a branch is reached, the structure of an execution trace typically resembles iterations of consecutive addresses followed by a branch target address (BTA). A BTA can be seen as a deviation from the consecutive flow of execution. The obvious redundancy of recording consecutive addresses can be eliminated if only BTAs are recorded along with a *length* denoting the number of consecutive instructions executed thereafter. Our first compression stage involves the elimination of consecutive addresses, as seen in Table 1, which also demonstrates why recording the length term is necessary. At the second BTA of the example, there are two possible execution paths to the next BTA. For the host to determine the path taken, it is necessary to also record the length.

Although the input to this stage is a 32-bit address, the output is the 30 MSBits of the address and an 8-bit length. On 32-bit fixed instruction-length processors, the 2 LSBits of an address are redundant and can be ignored. An 8-bit length term is used because our experimental benchmarks do not exceed a length of 255, as confirmed by us and [17].

3.2 Finite Context Method

Originally used in software-based trace compression, the Finite Context Method (FCM) [16] exploits sequential locality when sets of instructions are repeatedly executed. Based upon the n number of previously executed instructions, a prediction of the next instruction is made. The method operates similar to a cache, such that there is a *miss* the first time a set of instructions is encountered, and a *hit* for every subsequent encounter that matches the prediction.

Table 1: Example of CAE with 16-bit Addresses

Pre-CAE		Post-CAE	
Address	Instruction	BTA	Length
0xa120	add	0xa120	2
0xa122	sub		
0xa124	jmp 0xb144		
0xb144	load	0xb144	2
0xb146	sub	OR	
0xb148	bnz 0xc298	0xb144	4
0xb14c	xor		
0xb150	jmp 0xc298		
0xc298	jmp 0xd422	0xc298	0

In versions *v0* and *v1* of our implementation, depicted in Figure 3, a 16-bit *search key* is constructed by applying an exclusive-or (XOR) between staggered 16-bit portions of the last 4 instruction addresses (h1–h4) and the 4 LSBits of the length term (L). Versions *v2* and *v3* use the same scheme to construct a search key from 14 bits of each address, while *v4* uses 12 bits. By constructing the search key in this way, we aim to obtain a unique value when a set of address-length (AL) combinations is processed in a particular order. By including the length term in the key, loops where a branch is taken are made to generate different keys from ones where branches are not taken. In *v0* and *v1*, the key is used to address a 304 kB prediction table in SRAM which stores predicted AL combinations, each consisting of the 30 MSBits of the address concatenated with an 8-bit length. Versions *v2* and *v3* address 76 kB tables, while *v4* a 19 kB table. On every cycle, a new prediction is written to the table based upon the previous 4 input ALs, while a new prediction is simultaneously read for the newest AL. The storage of the prediction table in a dual-ported SRAM is essential to this operation. Both the predicted and input ALs are forwarded to the MTF/AE stage.

3.3 Move-to-Front & Address Encoding

This compression stage can be seen as an combination of two separate operations. The first implements a Move-to-Front (MTF) dictionary for ALs which are not correctly predicted at the FCM stage. The second encodes AL combinations as the smallest available representation before passing them to further compression stages.

Our MTF dictionary contains elements of 38-bits wide containing AL combinations, in depths of 256 elements for versions *v0* and *v1*, 128 elements for *v2* and *v3*, and 64 elements for *v3*. If the predicted AL does not match the input AL from the FCM stage, an MTF array search is initiated. Using parallel comparators, each element of the MTF array is compared with the input AL. If there is a match, the array index of the matching element is output. The element is then moved to the top of the array, and the remaining elements shifted down to take up its old position. If there is no MTF array match, the AL is placed at the top of the

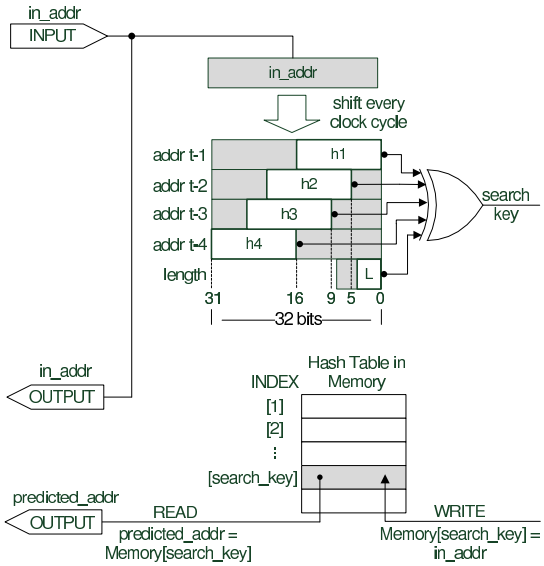

Figure 3: Finite Context Method – *Versions v0 & v1*

array, and the array shifted down by an element, thereby eliminating the least-recently-used (LRU) element.

Encoding of ALs is performed as per Table 2 by splitting them into two parts: a prefix byte and 1–5 data bytes. If the input AL matches the predicted AL, the output is a single 0x00 prefix byte, which represents a correct prediction. Otherwise, if the AL is located in the MTF array, the output is a 0x05 prefix byte along with the array index as the data byte. Failing both the previous possibilities, the size of the mispredicted AL is minimized by employing differential address encoding, which is demonstrated in Table 3 with 16-bit addresses (although 32-bit addresses are used in practice).

Differential address encoding also takes advantage of the fact that the addresses of a 32-bit word-aligned, fixed-length instruction processor have two redundant LSBits, just as a similar 16-bit processor would have a single redundant LSBit. Ignoring these bits, the differential encoding scheme compares the last input address with the current input address. From the 30-bit addresses, the scheme will output the minimum number of bytes of the input address that differ from the last address. Thus, the address could be encoded as an 8-bit, 16-bit, 24-bit, or 32-bit value, along with a prefix byte of 0x01, 0x02, 0x03, or 0x04, respectively. The prefix represents the number of *difference* address bytes to follow, which are in-turn followed by the single-byte length term.

Table 2: Address Encoding Scheme

Prefix Byte	Data Byte(s)	Meaning
00	–	correct prediction
01	addr[9:2] + len[7:0]	1 byte difference
02	addr[17:2] + len[7:0]	2 bytes from
03	addr[25:2] + len[7:0]	3 bytes previous
04	addr[31:2] + len[7:0]	4 bytes address
05	mtf_index[7:0]	found in MTF array

3.4 Data Stream Serializer

Since the *data* stream of the MTF/AE stage may reach up to 5 bytes in a single cycle, it must be serialized before being handed to the LZ stage. One solution is to use a small buffer which stores data from the MTF/AE stage, taking advantage of the fact that data output greater than 1 byte is infrequent. While this would typically suffice, for the sake of simplicity and robustness we feed the output into a serializer stage that is instead made to operate at 5x the clock frequency of the previous stages.

3.5 Run-length and Prefix Encoding

Of the possible prefix bytes in the MTF/AE stage (seen in Table 2), some can be observed to occur much more frequently than others. This property can be used to further compress the prefixes, by encoding the most frequently occurring prefixes with the shortest codes. In general, static encoding methods presume a fixed set of frequencies over time, and dynamic methods alter the encoding as frequencies change over time. Since we observed an overwhelming frequency pattern across a range of traces, the pre-computed static Huffman Tree seen in Figure 4 is used. The shortest single-bit code is assigned to a correct FCM prediction, which implies a probability of occurrence of 0.5. In addition, it can be observed that predictions are generally delivered in bursts. Rather than encoding each prediction as a single bit over a long span, a simple Run-Length Encoding (RLE) is applied to compress sequences of correct predictions.

The RLE operates by identifying sequences of up to 256 correct predictions and replacing them by a single byte. While there is no change to the prefix stream if there are 3 predictions or less, 4 or more will result in data output being paused, and a count performed on the predictions to follow. This will continue until the first non-prediction prefix is encountered, or the maximum count of 256 is reached. A byte containing the count will then be output. At best, this allows a sequence of 260 predictions to be represented by 12 bits (four single bits followed by a 1-byte count), and at worst also encodes a sequence of 4 predictions in 12-bits.

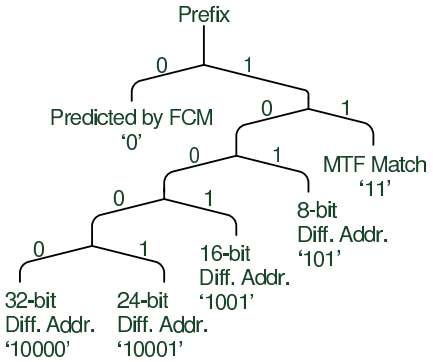

Figure 4: Huffman Tree for Prefix Encoding

3.6 Lempel-Ziv Encoding of Data Stream

The final compression stage is used to extract any non-obvious or vestigial redundancy that remains in the *data* stream of the MTF/AE stage. The input to this stage consists of MTF indices and differential AL combinations of 2–5 bytes. Since previous stages focused on compression at address-level granularity, applying a final compression stage at byte-level granularity can especially identify repeating combinations of input elements. For this purpose, the original Lempel-Ziv algorithm (LZ77) [18] is selected because it lends itself to a simple, yet effective hardware implementa-

Table 3: Example of Differential Address Encoding – 16-bit Addresses

Original Address	Original Address (binary)	Effective Address†	Effective Address (binary)†	Predicted by FCM/MTF?	Diff. from Prev. Addr.	Output Prefix	Output Data
0xa120	1010 0001 0010 0000	0x5090	101 0000 1001 0000	Yes	N/A	0x00	N/A
0xa144	1010 0001 0100 0100	0x50a2	101 0000 [1010 0010]*	No	1 byte	0x01	0xa2 + length
0xc298	1100 0010 1001 1000	0x614c	[110 0001 0100 1100]*	No	2 bytes	0x02	0x614c + length
0xc332	1100 0011 0011 0010	0x6199	110 0001 [1001 1001]*	No	1 byte	0x01	0x99 + length

†after ignoring the redundant LSBit of each 16-bit address, *boxed area denotes LSByte(s) that differ from the last input

Table 4: MiBench Compression Ratio Comparison

Benchmark	Instr. Count (mil.)	Compression Ratio								
		Others				Our Scheme (*s, m*)				
		gzip	bzip2	[17]	[14]	*v0* (16, 256)	*v1* (16, 128)	*v2* (14, 128)	*v3* (14, 64)	*v4* (12, 64)
blowfish_enc	781	94	322	113	99	144	127	127	94	91
crc32	111	215	1195	N/A	N/A	8662	8660	8660	8660	8660
fft	37	73	128	159	N/A	171	171	137	170	150
jpeg_comp	94	84	283	353	148	287	286	224	285	282
jpeg_decomp	23	47	176	613	189	448	446	444	426	416
sha	119	160	373	654	440	720	720	720	720	720
stringsearch	4	60	165	83	24	119	119	119	113	109
tiff2bw	158	93	386	2808	235	4803	4790	3280	4588	4212
tiff2rgba	191	75	503	5335	407	7651	7619	7614	7577	6817
tiffmedian	595	103	371	2713	414	2337	2336	2311	2303	2291
Overall	2113	81	257	236	181	322	290	289	224	217

tion. Even though compression is not guaranteed through this algorithm, in practice we found compression rates ranging from approximately -10% to 500% on the output of the MTF/AE stage. Our design operates on byte-wise elements, and implements a dictionary array of 256 elements. When an input byte is received, a parallel comparator searches the array for matching elements and flags all of those that match in a *tag* array consisting of 1 bit per element. The dictionary array is then shifted by 1 element and the input byte is added to the top. If there are flagged elements, then only they are searched as the next input is received. In this way, a sequence of matching bytes in the dictionary can be found by incrementally searching the array. When a sequence can no longer be matched, or if no match is initially found, a 3-byte output is issued that includes the original matching array index, the number of elements successfully matched (up to 256), and the new input byte.

4. EXPERIMENTAL RESULTS

The performance of our compression scheme was tested on applications of the MiBench [9] benchmark suite, which allowed a comparison to be made to the works of [17] and [14]. The suite includes programs that approximate typical office, telecommunications, security, and consumer workloads that have been used in the past to evaluate address trace compression performance. Execution traces were collected natively using an Apple PowerMac G4 with a 1.25 GHz PowerPC 7455, which is a 32-bit fixed instruction-length processor. Benchmarks were executed under Linux SMP kernel 2.6.32-24. Strictly for the purposes of data collection, a GDB script was used to record the PC register contents in ASCII. While this method of trace recording is relatively slow, it is accurate and portable to other architectures.

A VHDL description of the compression scheme was simulated in ModelSim SE-64 v6.5c. Results were verified by using a decompression program executed in MATLAB R2010a to decompress each trace and compare it to the original.

In Table 4, all versions of our compression scheme are compared with general-purpose software compressors *gzip* v1.3.12 and *bzip2* v1.0.5, as well as with the works of [17] and [14]. Compression ratios are derived by dividing the size of the original trace by the size of the compressed trace.

Our scheme can be seen to generally outperform the others, aside from the *tiffmedian* and *jpeg* applications. The final row of the table represents the average performance in a mixed-execution scenario where all included benchmarks are run in a sequence. Our version *v0* represents an overall compression performance improvement of 44% over [14] and 27% over [17], while outperforming both general-purpose algorithms. Likewise, versions *v1–v4* are able to maintain strong compression ratios with much less memory usage than *v0*. All versions outperform the commercial tools ETM and MCDS, which are reported to reach up to 80x [2] and 20x [12] compression on unspecified workloads, respectively. In general, the use of differing benchmarks between schemes in the literature makes it difficult to directly compare compression performance.

To gauge the size of the design on an FPGA, hardware synthesis was performed in Altera Quartus II v9.1. Logic utilization for every version of the scheme, including a per-stage breakdown for *v0*, can be seen in Table 5 for an Altera Stratix III (EP3SL200F1152C2) device. Effective emulation speed would match reported clock rates if a small buffer is used in the Serializer stage, or would be 1/5th the reported speeds if the Serializer and LZ stages are instead clocked at 5x the main clock. The trade-off between clock speed and

MTF array size is apparent due to the structure of the long MTF shift register. The schemes represent between 2–5% of available ALUT resources for this device, and between 3–8% of available logic registers.

Table 5: Altera Stratix III Logic Utilization

Version (s, m)	ALUTs	Regs.	Memory (kB)	Speed (MHz)
v0 (16, 256)	7859	12597	304	68
CAE	65	108	–	261
FCM	63	146	304	325
MTF/AE	5305	9808	–	75
Ser.	155	113	–	275
RL	112	86	–	277
LZ77	2183	2369	–	119
v1 (16, 128)	5210	7735	304	99
v2 (14, 128)	5170	7724	76	102
v3 (14, 64)	3830	5285	76	131
v4 (12, 64)	3839	5289	19	139

4.1 Usage Scenario

The ubiquitous IEEE 1149.1 (JTAG) interface is widely used for software debugging, but is limited to a clock speed of approximately 100 MHz, which allows a maximum theoretical transfer rate of 12.5 MB/s. Consider the example from Section 1, of a 32-bit emulated processor capable of 1 CPI and clocked at 100 MHz, which produces 400 MB/s of address trace data. Without trace compression, the system must be clocked at 3 MHz to enable a single thread of address trace data to be outputted. With the compression ratio of our version *v2*, emulating the SoC at full speed only requires an average of 1.38 MB/s, which is well within the bandwidth of the JTAG port. Given a method of interleaving between cores, it allows at-speed simultaneous emulation of up to 9 cores over a single JTAG link. In a real MPSoC, version *v0* would likewise allow up to a 1 GHz, 1 CPI core to be traced via JTAG.

Assuming the probability of a software fault f is $P(f) = 10^{-13}$, at 100 MHz it would take an average of 28 hours for a single fault to manifest itself. In our debug scenario, a continuous trace could be collected for this duration without slowing down emulation speed. Without trace compression, it would require an average of 39 days for the fault to appear.

5. CONCLUSION AND FUTURE WORK

In this paper we have presented a parameterizable microarchitecture for address trace compression, suited to implementation on ASICs and modern FPGAs. Our method offers better overall performance than other schemes by exploiting higher-order redundancy in address trace streams. The design occupies modest area by harnessing unused embedded memory on emulation platforms and the trace buffer typically found on CPUs. The result is a dramatically smaller volume of trace data through the debug-link, allowing real-time address tracing of a 1 GHz processor or 9 emulated FPGA cores via a single JTAG link. As a result, software debugging can be accelerated by an order of magnitude over uncompressed trace extraction. In future, we plan to combine our software debugging efforts with assertion-based debugging [3], as part of a testing, monitoring, and debugging infrastructure for SoCs and networks-on-chip [4].

6. REFERENCES

[1] E. Anis and N. Nicolici. On using lossless compression of debug data in embedded logic analysis. In *IEEE International Test Conference, 2007*, pages 1–10, 2007.

[2] ARM Ltd. CoreSight Trace Macrocells. http://www.arm.com/products/system-ip.

[3] M. Boulé, J. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *International Conference on Computer Design*, pages 294–299, 2006.

[4] S. Bourduas, J. Chenard, and Z. Zilic. A Quality-Driven design approach for NoCs. *IEEE Design & Test of Computers*, 25(5):416–428, 2008.

[5] M. Burtscher, I. Ganusov, S. Jackson, J. Ke, P. Ratanaworabhan, and N. Sam. The VPC trace-compression algorithms. *IEEE Transactions on Computers*, 54(11):1329–1344, 2005.

[6] E. Chung and J. Hoe. High-Level design and validation of the BlueSPARC multithreaded processor. *IEEE Transactions on CAD*, 29(10):1459–1470, 2010.

[7] E. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai. A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs. In *Proceedings of the International Symposium on FPGAs*, pages 77–86, 2008.

[8] P. A. Emrath, S. Chosh, and D. A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing. ACM/IEEE Conference on*, pages 580–588, 1989.

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: a free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[10] A. Hopkins and K. McDonald-Maier. Debug support for complex systems on-chip: a review. *IEE Proc. of Computers & Digital Techniques*, 153(4):197–207, 2006.

[11] C. Kao, S. Huang, and I. Huang. A hardware approach to Real-Time program trace compression for embedded processors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(3):530–543, 2007.

[12] A. Mayer, H. Siebert, and C. Lipsky. Multi-core debug solution IP. White paper, IPExtreme, 2007.

[13] A. Mayer, H. Siebert, and K. McDonald-Maier. Boosting debugging support for complex systems on chip. *Computer*, 40(4):76–81, 2007.

[14] M. Milenkovic and M. Burtscher. Algorithms and hardware structures for unobtrusive real-time compression of instruction and data address traces. In *Data Compression Conference*, pages 283–292, 2007.

[15] B. Plattner. Real-Time execution monitoring. *IEEE Trans. Software Engineering*, SE-10(6):756–764, 1984.

[16] Y. Sazeides and J. Smith. The predictability of data values. In *Microarchitecture., IEEE/ACM International Symposium on*, pages 248–258, 1997.

[17] V. Uzelac and A. Milenkovic. A real-time program trace compressor utilizing double move-to-front method. In *DAC'09*, pages 738–743, 2009.

[18] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.