A Real-Time Program Trace Compressor Utilizing Double Move-to-Front Method

Vladimir Uzelac, Aleksandar Milenkovic
The University of Alabama in Huntsville
{uzelacv, milenka}@ece.uah.edu

ABSTRACT
This paper introduces a new unobtrusive and cost-effective method for the capture and compression of program execution traces in real-time, which is based on a double move-to-front transformation. We explore its effectiveness and describe a cost-effective hardware implementation. The proposed trace compressor requires only 0.12 bits per instruction of trace port bandwidth, at the cost of 25K gates.

Categories and Subject Descriptors
B 7.2 [Integrated Circuits]: Design Aids-Verification. D.2.5: [**Testing and Debugging**]: Debugging aids, Tracing.

General Terms
Design, Verification.

Keywords
Debugging, Program Trace, Compression.

1. INTRODUCTION
Continual growth in the complexity of embedded systems-on-a-chip (SoCs) makes traditional approaches to system-level testing and debugging infeasible or impractical. For example, the development of a dedicated In-Circuit-Emulator (ICE) with additional support for debugging is cost-prohibitive; in addition, the ICE's physical characteristics such as chip floorplan, pin layout, and timing characteristics, differ from the targeted SoC. Traditional software approaches to debugging that rely on hardware and software breakpoints are often insufficient to capture the real sources of a bug. Moreover, they interfere with normal program execution, often causing the original error to disappear. This is especially important for real-time and safety-critical embedded systems that often need to be tested in real operating conditions. Last but not least, software step-by-step debugging is time consuming and places an additional strain on system developers, resulting in either poorly tested designs or product delays or both.

Embedded processor manufacturers responded to this debugging challenge by incorporating on-chip hardware resources exclusively dedicated to program tracing and debugging. For example, ARM based embedded systems may include Embedded Trace Macrocell [1] modules to support program tracing; Altera Nios II [2] and Xilinx Microblaze [3] based systems may also include trace modules to enable real-time tracing of programs and data. Lauterbach [4] offers a number of program tracing hardware and software tools for a variety of processors. Typically, trace modules capture instruction and data traces (and possibly other bus signals), perform branch filtering, and store traces in on-chip trace buffers. The trace buffers can be read by external trace units through a JTAG interface or through the system bus. Alternatively, a trace module can send traced data directly through a trace port. The traces are then used in conjunction with program binaries to faithfully replay program execution and locate a bug source. In addition to debugging, program execution traces are also vital for workload characterization and performance tuning and optimization.

The existing commercially available trace modules rely either on large on-chip buffers to store execution traces of sufficiently large program segments or on wide trace ports that can sustain a large amount of trace data in real-time. However, large trace buffers and/or wide trace ports significantly increase the system complexity and cost. Moreover, they do not scale well, which is a significant problem in the era of multicore systems.

Compressing program execution traces at runtime in hardware can be used to reduce requirements for on-chip trace buffers and trace port communication bandwidth. Whereas commercially available trace modules typically implement only rudimentary forms of hardware compression with a relatively small compression ratio (5:1), several recent research efforts in academia propose effective trace compression techniques that can achieve compression ratios one order of magnitude higher [5-7]. For example, Kao *et al*. [5] propose an LZ-based program trace compressor that achieves a good compression ratio for a selected set of programs. However, the proposed module has a relatively high complexity (50K gates). In addition, the selected program segments are relatively small with less than 10 million instructions, so it is unclear how effective it would be in tracing more diverse programs.

In this paper we introduce a new cost-effective technique for compression of program traces in real-time. The proposed technique exploits common program characteristics and utilizes a two-level move-to-front transformation. We thoroughly explore program characteristics with regard to trace compression (Section 2), introduce a new Double Move-To-Front method (DMTF) for compression of program traces (Section 3), explore its design space (Section 4), and describe a cost-effective hardware implementation (Section 5). We also introduce two enhancements to the original method and explore their effectiveness using 17 diverse benchmarks from the MiBench benchmark suite [8]. A

DAC'09, July 26-31, 2009, San Francisco, California, USA

trace module configuration of 25,000 logic gates achieves compression ratios between 83 and 29,389, depending on the benchmark. The average weighted compression ratio is 268:1, which translates into 0.12 bits/instruction.

2. PROGRAM CHARACTERISTICS AND MOVE-TO-FRONT TRANSFORMATION

To replay a program flow offline, we only need to trace the information about program dynamic basic blocks (or streams). An instruction stream is a sequential run of instructions, from the target of a taken branch to the first taken branch in the sequence. Each instruction stream can be uniquely represented by its starting address (SA) and its length (SL). Thus, the complete trace of instruction addresses from an instruction stream can be replaced by the corresponding stream descriptor, i.e., the (SA, SL) pair. Relatively simple logic can be used to capture (SA, SL) pairs. In processors with fixed instruction word length, the current program counter (PC) is compared to the previous PC. If they differ for a value other than the instruction length, the current instruction is the beginning of a new stream. The current values of the SA and SL registers are output and the current PC is moved to the SA register to mark the beginning of a new stream. The SL register is set to 1. If the difference corresponds to the instruction length, the current value in the SL is incremented. In processors with variable instruction length, the stream detector requires an additional control line from the CPU to indicate a taken branch instruction. We introduce a slight modification to the original definition of an instruction stream. When we encounter an unconditional direct branch we do not terminate the current stream because the address of the next instruction in sequence can be inferred directly from the binary. Thus, when such a branch is identified, the SL register is just incremented as it was a non-branching instruction.

Most programs have only a small number of unique program streams, with just a fraction of them responsible for majority of program execution. Figure 1 shows some important characteristics of MiBench [8] benchmarks collected using SimpleScalar [9] while running ARM binaries. The first 4 columns *(a-d)* show the number of executed instructions (in millions), the number of unique streams, the maximum and average stream length, respectively. The number of unique streams ranges from 341 to 6871, and the average dynamic stream length is between 5.9 (*bf_e*) and 54.7 (*adpcm_c*) instructions. The fifth column *(e)* shows the number of unique program streams that constitute 90% of dynamically executed streams. This number ranges between 1 and 235, and it is 78 on average. Note that all calculations assume weighted average, where weights are determined based on the number of executed instructions, since the raw instruction address trace is directly proportional to the number of executed instructions. The maximum stream length never exceeds 256, thus we may choose to use 8 bits to represent SL. In addition to this, it can be shown that these frequently executed program streams create repeating patterns with strong local correlation. Our approach is to exploit these program characteristics in designing a cost-effective trace compressor that will achieve an excellent compression ratio with minimal storage and trace port bandwidth requirements.

Move-to-Front (MTF) [10] is an encoding of data designed to reduce the entropy of symbols in a data message by exploiting the local correlation between symbols. It is used in conjunction with the Burrows-Wheeler transform in the *bzip2* utility program [11].

The MTF algorithm encodes an input data message as follows. If an incoming input symbol is found in a history table *ht*, it is replaced with its index *i* in the *ht*, and the symbol is moved at the top of the table (the entry with index 0). The *ht* is updated by shifting down first *i-1* entries by one position, such that *ht*[*i*]=*ht*[*i*-1], ..., *ht*[1]=*ht*[0]. To illustrate the MTF operation, let us consider an input message *AABC*, and a history table *ht*=[*C*, *B*, *A*] (symbol *C* is at the position 0). The MTF transforms the 3-symbol input message into a new 2-symbol message *2022*.

	IC (mil.)	SC	Max SL	Avg SL	CDF 90%	ht CDF 90%	ht[x] HR	ht2[0] HR
adpcm_c	733	341	71	54.7	1	1	0.99	1.00
bf_e	544	403	70	5.9	22	5	0.41	0.69
cjpeg	105	1590	239	12.3	47	11	0.46	0.90
djpeg	23	1261	206	25.1	31	11	0.69	0.87
fft	631	846	94	10.5	209	32	0.17	0.66
ghostscript	708	6871	251	10.0	67	22	0.20	0.76
gsm_d	1299	711	165	19.5	33	6	0.48	0.90
lame	1285	3229	237	32.4	235	21	0.23	0.74
mad	287	1528	206	20.7	42	25	0.30	0.75
rijndael_e	320	513	77	21.0	45	2	0.57	0.79
rsynth	825	1238	180	17.6	49	10	0.26	0.77
stringsearch	4	436	65	6.0	48	38	0.62	0.81
sha	141	519	65	15.4	10	2	0.86	0.92
tiff2bw	143	1038	43	12.8	2	1	0.97	0.99
tiff2rgba	152	1131	75	27.7	2	1	0.92	0.99
tiffmedian	541	1335	92	22.3	5	1	0.91	0.97
tiffdither	833	1777	67	14.3	63	38	0.45	0.78
Average	**816**	**1791**	**145**	**21.6**	**77.8**	**14.5**	**0.46**	**0.82**
	(a)	*(b)*	*(c)*	*(d)*	*(e)*	*(f)*	*(g)*	*(h)*

Figure 1. MiBench program characteristics.

The original MTF transformation can be easily extended to allow operation starting from an empty history table. The history table is searched for an incoming input symbol. If the symbol is not found in the table (we call this event an *ht* miss), the original symbol is output and the table is updated by shifting its content by one position and by placing the incoming symbol in the *ht*[0]. If the symbol is found in the history table (an *ht* hit event), its index is output and the table is updated as described above. The MTF allows for an effective encoding of frequently executed program sections. Let us consider several typical examples of program loops where symbols A, B, and C represent unique instruction streams characterized by their respective (SA, SL) pairs. For example, a program loop consisting of a sequence of two streams A and B repeating many times, illustrated as {AB}, is transformed into a hit pattern {11}; similarly, a loop with a repeating pattern {ABC} is transformed into {222}.

A relatively small history table will suffice to achieve a good hit rate due to a strong temporal locality of instruction streams in common programs. When a stream descriptor (SA, SL) is found in the history table, it is replaced with its index in the history table. Otherwise, the full stream descriptor of 40 bits is output in case that the SA is a target of an indirect branch. If the SA is a target of a direct branch, it can be inferred from the program binary, and we output only 8 bits for SL. The effectiveness of the MTF transformation on program execution traces consisting of a sequence of stream descriptors is shown in Figure 1*(f)*. We measure the frequency of the output symbols after the MTF transformation is applied. The average number of unique MTF output symbols that constitute 90% of all dynamically executed program streams is only 14.5 (down from 78 before the MTF transformation), ranging from 1 (*adpcm_c*) to 38 (*stringsearch*).

Note: the experiments are conducted assuming a history table with 128 entries; the hit rate is over 97%, so a very small number of streams are not transformed with the MTF.

A perfect trace compression without stream pattern recognition would replace each stream with just a single bit. As described above, the MTF transformation significantly reduces the number of trace symbols. To come close to a one bit per stream goal, we need to identify the most frequent entry and to encode it with a single bit. Figure 1*(g)* shows the percentage of the hit events in the most frequent *ht* entry. Although this percentage is fairly high for many benchmarks (e.g., *adpcm_c*, *tiff2bw*), it is relatively modest for others (e.g, 17% for *fft*, and 46% on average across all benchmarks). An additional problem is how to identify the most frequent entry in the *ht* because it varies across benchmarks.

In order to resolve these two problems, we introduce an additional, second level move-to-front transformation. Let us consider the following repeating stream pattern {ABAC}. The hit pattern at the output from the first-level MTF is {1212}. If we supply this pattern to the second-level MTF history table, the hit pattern at the output is {1111}, with even lower entropy of information. Because the MTF transformation lowers the number of frequent symbols, the level 2 history table can be significantly smaller.

This approach can be further extended by introducing another level of MTF transformation; in general we could introduce a hierarchy of MTF history tables. The size of the MTF history tables will exponentially decrease as we move toward the upper levels. However, an increase in the number of MTF levels will reach the point of diminishing returns, and will not yield expected gains. In general, the optimal configuration is application specific. Our analysis shows that a 2-level MTF configuration appears to be optimal. Figure 1*(h)* shows a high percentage of program streams that end up in the entry 0 of the level 2 history table (*ht2*), from 66% to 100%. By encoding this entry with a single bit we approach the goal of having one bit per instruction stream. Note: the experiments are conducted using *ht2* with 16 entries achieving 94% hit rate.

3. DMTF METHOD

The analysis from the previous section suggests the use of a 2-level move-to-front transformation as optimal in compressing program instruction traces. Consequently, we design an instruction trace compressor with two history tables in sequence. We name this scheme Double Move-to-Front (DMTF). The first- and the second-level history tables are named *mtf1* and *mtf2*, respectively.

Figure 2 illustrates operation of the proposed trace compressor. When a new stream is detected, its descriptor (SA, SL) is forwarded to *mtf1*. As described before, the *mtf1* table is searched for the stream descriptor. If we find a match, we have an *mtf1* hit; the index of the matching entry is output to the next stage, and *mtf1* is updated accordingly. Otherwise, we have an *mtf1* miss; the *mtf1* content is shifted down by one position, and *mtf1[0]* is loaded with the stream descriptor. In case of an *mtf1* hit, the index *i1* is sent to the *mtf2* history table; *mtf2* is searched for the index *i1*. If we find a match in the entry 0, *mtf2*[0], we have an *mtf2* zero entry hit. If we find a match in the remaining *mtf2* entries, we have an *mtf2* non-zero entry hit. Otherwise we have an *mtf2* miss event.

We can distinguish four different events in the DMTF scheme and they are encoded as follows. An *mtf2[0]* hit is encoded with a single bit '0'. An *mtf2* non-zero entry hit is encoded with a one-bit header '1' and the *mtf2* index *i2* ('1'+*i2*). An *mtf1* hit with a miss in *mtf2* is encoded with ('1'+i2miss+*i1*); note that the last index in the *mtf2* table, *i2miss*, is reserved to indicate a miss event in the *mtf2*. Finally, a miss in *mtf1* is encoded with a header ('1'+i2miss+i1miss) followed by a full or a partial stream descriptor ([SA], SL) – 40 or 8 bits. Note: the last index in the *mtf1* table, i1miss, is reserved to indicate a miss in the *mtf1*.

The compression ratio that can be achieved using DMTF scheme can be expressed analytically as follows. Equation 1 shows the number of bits needed to encode a single stream after DMTF compression, as a function of five parameters: *mtf2* zero-entry hit rate, *mtf2.zhr*; *mtf2* non-zero entry hit rate, *mtf2.ohr*; *mtf1* hit rate, *mtf1.hr*; *mtf1* size, *mtf1.size*; and *mtf2* size, *mtf2.size*. Note: *mtf2.hr=mtf2.zhr + mtf2.ohr*. Equation 2 shows the compression ratio as a function of the number of instructions in a program, the number of executed instruction streams, and the number of bits per one instruction stream.

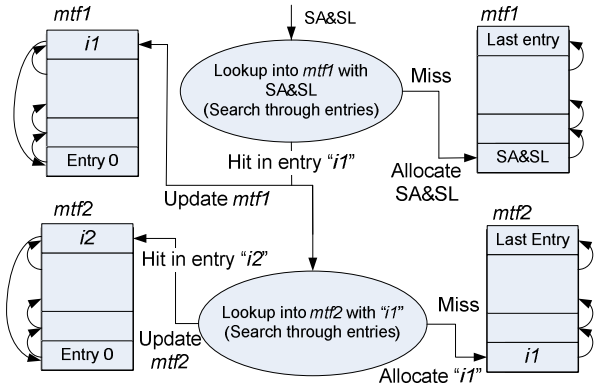

Figure 2. DMTF Operation.

Eq. 1
$$\begin{aligned} BitsPerStream = {} & mtf2.zhr + \\ & (mtf2.ohr * (1 + \log_2(mtf2.size)) + \\ & (mtf1.hr - mtf2.hr) * (1 + \log_2 mtf2.size + \log_2 mtf1.size) + \\ & (1 - mtf1.hr) * (1 + \log_2 mtf2.size + \log_2 mtf1.size + 8 + [32]) \end{aligned}$$

Eq. 2
$$CR = \frac{32 * InstructionCount / StreamCount}{BitsPerStream}$$

An Example Compression/Decompression. Let us illustrate the compression flow using an example from Figure 3*(a)*. We consider the following sequence of instruction streams *ABCAABABAC*, where A, B, and C denote 3 instruction streams with distinct stream descriptors. Let us assume a 64-entry *mtf1* and an 8-entry *mtf2*. Note that the actual number of entries is 63 and 7, respectively, since the last indices are reserved to indicate miss events. First three instruction streams are not found in the *mtf1* and are output with the header '1', followed by a 3-bit index in the *mtf2* reserved for miss events ('111'), a 6-bit index in the *mtf1* reserved for miss events ('111111'), and individual stream descriptors ([SA], SL). Next, the stream A is found in *mtf1*[2], but index 2 is not found in the *mtf2* resulting in an *mtf2* miss with *mtf1* hit event; we emit a header '1'+'111' followed by the *mtf1* index

'000010'. The next stream in sequence is A, resulting again in an *mtf2* miss with *mtf1* hit event; this event is encoded with '1+'111'+'000000'. The rest of the compression flow continues as illustrated in Figure 3*(a)*.

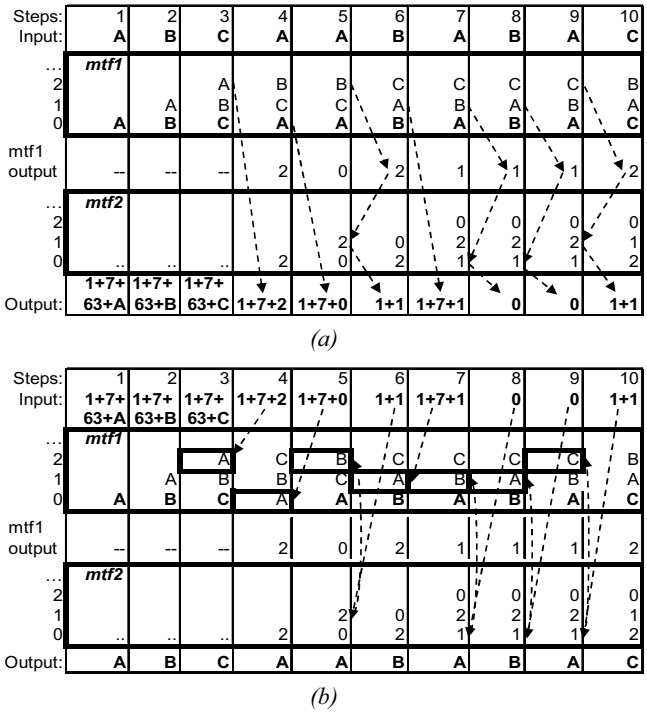

Figure 3. DMTF compression (a) and decompression (b) flow examples.

The decompression flow is a reversed compression flow and it requires the same configuration of the history tables. The compressed trace is read, headers are analyzed and the history tables updated according to the DMTF method described above. The decompression flow is illustrated in Figure 3*(b)*. The first item starts with the header '1'+'111'+'111111', which indicates that the next 40 bits represent the first stream descriptor, (SA, SL). The stream descriptor is loaded into *mtf1*[0]. The de-compressor now can recreate a complete instruction trace for this stream. The next two items in the trace are decompressed in the same way. The next trace record '1+'111'+'000010' directs the de-compressor to find the original stream descriptor in *mtf1*[2] (instruction stream A). The following trace record '1'+'111'+'000000' directs the de-compressor to find the stream in *mtf1*[0], which is stream A. The rest of the decompression process is illustrated in Figure 3*(b)*.

Performance Analysis. Figure 4*(a)* shows the compression ratio for MiBench benchmarks; the size of the *mtf1* is fixed to 128 entries and the *mtf2* size is varied from 4 to 16 entries. The last row (*Average*) shows the total compression ratio calculated as the weighted harmonic mean of individual benchmark compression ratios. The results show that we are able to achieve an excellent compression ratio ranging from 45 to 1738. The results also indicate that a DMTF configuration with only 4-entry *mtf2* will outperform configurations with larger *mtf2*.

	CR (mtf1 size = 128)			*Distribution per component*			
mtf2 size	*4*	*8*	*16*	*zht*	*mtf2ht*	*mtf1ht*	*mtf1mt*
adpcm_c	1738	1734	1729	99%	1%	0%	0%
bf_e	110	94	84	40%	52%	8%	0%
cjpeg	250	253	252	57%	10%	31%	2%
djpeg	434	434	433	47%	12%	30%	11%
fft	45	44	43	9%	3%	10%	78%
ghostscript	107	106	107	25%	7%	55%	13%
gsm_d	358	343	327	52%	13%	4%	32%
lame	326	329	323	23%	12%	28%	37%
mad	210	210	212	24%	8%	48%	20%
rijndael_e	323	409	362	38%	17%	45%	0%
rsynth	209	199	186	29%	18%	13%	41%
strings.	81	78	75	34%	7%	58%	2%
sha	425	399	375	79%	20%	0%	0%
tiff2bw	391	390	388	95%	2%	3%	0%
tiff2rgba	852	844	834	95%	3%	0%	1%
tiffmedian	523	515	505	71%	5%	2%	22%
tiffdither	170	161	155	29%	9%	44%	18%
Average	**181**	**175**	**169**	**42.8%**	**12.3%**	**20.0%**	**25.0%**

(a) *(b)*

Figure 4. Compression ratio for DMTF(128,*X*), *X*= 4-16 *(a)*. Distribution of the individual trace components *(b)*.

4. ENHANCED DMTF METHOD

The output of the DMTF trace compressor contains a lot of redundant information. We introduce two low-cost enhancements that exploit this redundancy and/or reduce complexity of the compressor implementation. The four components of the output trace, *mtf2* zero hit trace (*zht*), *mtf2* non-zero hit trace (*mtf2ht*), *mtf2* miss with *mtf1* hit trace (*mtf1ht*), and *mtf1* miss trace (*mtf1mt*) are analyzed separately. Figure 4*(b)* shows the contribution of each component to the total trace size for DMTF(128,4) (128-entry *mtf1* and 4-entry *mtf2*). The results show the *mtf1mt* component is responsible for 25% of the total size, in spite of high hit rates in the *mtf1*. Fortunately, the redundant information in this trace can be easily exploited using a simple last-value predictor on upper address bits that stay constant during program execution. This enhancement is described in Section 4.1 and also helps reduce hardware complexity of the compressor implementation. Next, the zero trace occupies 43% of the total trace. We expect it to contain long runs of '0's, and its size can be reduced by replacing them by a counter value (Section 4.2). Finally, in Section 4.3 we put both enhancements together and evaluate the effectiveness of the DMTF instruction trace compressor.

4.1 Last-Value Predictor for Upper Address Bits

The upper address bits of the starting address (SA) field in the stream descriptor rarely change during program execution. We analyzed the locality of stream starting addresses; the SA field of the incoming stream is compared bit by bit to the SA of the previous instruction stream or to SAs of the several last instruction streams. The results indicate that the upper 12 address bits, SA[31:20], stay constant during program execution in 99% of cases. Therefore, we divide the SA field into two parts: the lower 20 address bits SA[19:0] that are compressed through the DMTF, and the upper 12 bits that are handled using a simple last value predictor (*HLV*). Note: SA[1:0] is '00' for the ARM ISA and could be omitted; SA[0]= '0' for the ARM Thumb ISA, so the SA[0] could be omitted. Here we keep the whole address.

A 12-bit last value (LV) register keeps the upper 12 bits of the last stream's SA. The upper 12 address bits of an incoming stream are compared to the LV. If they match we have an *HLV* hit. The

lower 20 address bits SA[19:0] and SL are used in *mtf1* lookup. We adopt a scheme where *mtf1* hits are conditional upon the corresponding *HLV* hits. An *HLV* miss will cause that a miss *trace* record is emitted regardless of mtf1 hits. When we have an *HLV* hit with *mtf1* miss event, the upper address bits are not emitted (in case that a full stream descriptor is required). Finally, in case when both *mtf1* and *HLV* have a hit, a regular DMTF record is emitted. The miss trace format is consequently extended to support these modifications.

The effectiveness of this enhancement is analyzed below. In general, it is beneficial in DMTF configurations with a relatively small *mtf1* and less so with a larger *mtf1*. The performance benefits are somewhat limited because direct branches dominate in the MiBench suite (92% of all branches on average) and all stream descriptors that start with targets of direct branches do not require the SA field. However, it significantly reduces the complexity of the DMTF implementation, as we do not need to keep upper 12 address bits in the *mtf1* history table.

4.2 Zero Hit Trace Counters

We show that the DMTF method ensures that the *mtf2* zero hit event is the most frequent one, and thus it is encoded with a single bit '0'. In many benchmarks the output trace will consist of long runs of zeros. The redundancy in this trace can be exploited by utilizing a zero-length counter (ZLC for short); it counts the consecutive zeros and replaces them with a counter value preceded by a new header. The number of bits used to encode this trace component is determined by the counter size. A longer counter can capture longer runs of zeros, but too long counter results in wasted bits. Our analysis of the *zht* trace component shows a fairly large variation in the average number of consecutive zeros, ranging from 5 in *ghostscript* and *fft* to hundreds of in *adpcm_c* and *tiff2bw*. In addition, zero runs in a program may vary across different program phases. This implies that an adaptive ZLC length method would be optimal.

The adaptive zero-length counter (AZLC) tries to dynamically adjust the ZLC size to the program flow characteristics. An additional 4-bit saturating counter monitors the *zht* component and it is updated as follows. It is incremented by 3 when the number of consecutive zeros in the trace (*mtf2*[0] hits) exceeds the current size of the ZLC. The monitoring counter is decremented by 1 when a detected zero sequence is smaller than the ZLC counter maximum value. When the monitoring counter reaches the maximum (15) or minimum (0) values, a change in the ZLC size occurs.

The AZLC requires a slight modification of the trace output format. A header bit '0' is followed by $\log_2$(AZLC Size) bits. The counter size is automatically adjusted as described above. The decompressor needs to implement the same adaptive algorithm.

4.3 Putting It All Together

Figure 5 shows a modified trace format that supports two enhancements, HLV and AZLC. Figure 6 shows the average compression ratio (CR) of several DMTF configurations as a function of the *mtf1* size (64-320). The basic DMTF (bDMTF) with *mtf2*=4 performs better than with *mtf2*=8 for any *mtf1* size as previously indicated in Figure 4. The DMTF with HLV predictor and *mtf2*=4 (hDMTF) performs better than bDMTF only for small *mtf1* sizes. When *mtf1*=192 bDMTF slightly outperforms hDMTH primarily due to significant performance degradation for lame benchmark. Finally, the enhanced DMTF with *mtf2*=4 (eDMTF) with both improvements performs the best. For all configurations the compression ratio saturates for the *mtf1* with 256 entries, and the *mtf1* with 192 entries strikes an optimal balance between the complexity and compression ratio. Figure 6 gives a design guideline and shows how one can trade compression ratio for complexity (the most complex resource in the enhanced DMTF module is the *mtf1* table).

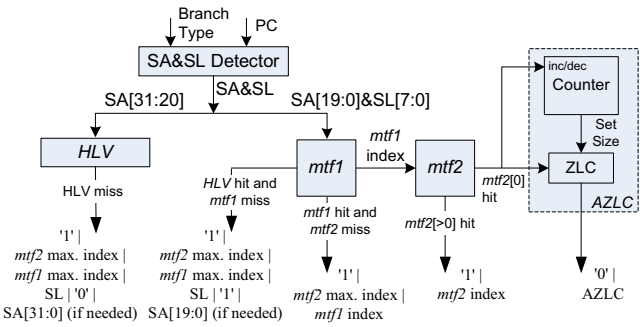

Figure 5. An Enhanced DMTF Trace Format.

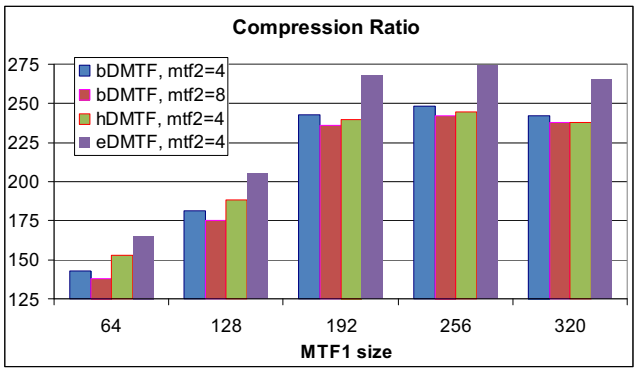

Figure 6. Compression ratio as a function of the *mtf1* size.

	mtf1=192, mtf2=4				*mtf1=64, mtf2=4*			
	CR bDMTF	*CR hDMTF*	*CR eDMTF*	*bits/inst. eDMTF*	*CR bDMTF*	*CR hDMTF*	*CR eDMTF*	*bits/inst. eDMTF*
adpcm_c	1738	1738	29389	0.001	1738	1738	29441	0.001
bf_e	108.8	108.8	112.7	0.284	110.5	110.5	114.5	0.279
cjpeg	245.3	245.7	353.1	0.091	245.6	248.7	360.5	0.089
djpeg	448.9	451.6	612.9	0.052	421.9	433.6	582.4	0.055
fft	151.7	151.9	159.1	0.201	26.9	31.1	31.6	1.012
ghostscript	104.7	106.1	104.6	0.306	104.8	108.4	106.9	0.299
gsm_d	495.4	495.4	808.4	0.040	363.2	381.2	547.8	0.058
lame	317.1	274.2	283.2	0.113	318.6	279.7	289.8	0.110
mad	202.7	208.8	217.0	0.147	219.0	227.6	237.4	0.135
rijndael_e	308.9	308.9	333.5	0.096	333.7	334.5	363.4	0.088
rsynth	307.2	307.4	296.3	0.108	159.1	176.6	173.8	0.184
strings.	77.0	77.2	82.7	0.387	32.9	37.2	38.8	0.825
sha	424.9	425.1	654.3	0.049	425.1	425.2	654.7	0.049
tiff2bw	390.2	390.4	2807.6	0.011	221.3	241.0	521.8	0.061
tiff2rgba	852.5	853.8	5334.6	0.006	348.8	393.5	637.7	0.050
tiffmedian	653.1	653.4	2712.9	0.012	386.1	418.7	819.1	0.039
tiffdither	167.8	169.6	193.2	0.166	140.9	144.8	162.7	0.197
Average	**242.5**	**239.5**	**268.1**	**0.119**	**143.1**	**153.1**	**165.1**	**0.193**

Figure 7. Compression ratios for xDMTF (x=b,h,e).

Figure 7 shows a detailed evaluation for bDMTF, hDMTF, and eDMTF with two configurations (192,4) and (64,4), x={b,h,e}. The hDMTF configuration achieves 7% higher CR than bDMTF for *mtf1*=64. This improvement is due to reducing the size of the miss trace. For *mtf1*=192, we see a decrease in hDMTF

performance over bDMTF as explained above. The eDMTF configuration achieves 15% higher CR over bDMTF for *mtf1*=64 and 11% for *mtf1*=192. This improvement is unevenly distributed over benchmarks and is useful for tests such as *adpcm*_c (17x) or *tiff2rgba* (6x). The best performing configuration (eDMTF with *mtf1*=192) achieves the total weighted average bandwidth on the trace port of only 0.12 bits per instruction.

5. DMTF HARDWARE IMPLEMENTATION

The *mtf1* and *mtf2* history tables can be implemented as custom fully associative structures with a single-clock cycle lookup and additional hardware needed to support the move-to-front update operation. Instead, we propose a cost-effective implementation that combines a standard content addressable memory (CAM) and a most-recently used (MRU) stack (Figure 8). The MRU stack has the same number of entries as the history table, but its content are indices in the CAM memory. Each MRU stack entry points to a particular CAM location, and thus has [$\log_2$(MTF_Size)] bits.

The *mtf* lookup operation encompasses a lookup into the CAM with (SA, SL) pair and a lookup into the MRU stack. In case of a CAM hit, the corresponding CAM index is forwarded to the MRU stack and the MRU lookup is performed. The selected entry is moved at the top of the MRU stack, and the top *(i-1)* locations are shifted down. In case of a CAM miss, the MRU stack provides the address of the CAM location where the new stream is going to be stored (the index at the bottom of the MRU stack), and the MRU stack is updated accordingly. Figure 8 shows a block diagram of a single level MTF history buffer. The lookup and update together require only two processor clock cycles and are performed only when a new instruction stream is detected. Hence, the compression can be done at the full processor speed without ever slowing the processor.

To estimate the complexity of the proposed implementation we consider enhanced DMTF(192,4) configuration. The *mtf1* CAM memory has 191 entries, each with 28 bits (20 for SA, and 8 for SL). With 3 gates per CAM bit [12], the CAM complexity is estimated at 3x28x191 ~ 16000 logic gates. The *mtf1* MRU stack has 191 8-bit entries, plus comparators attached to each of them. Registers use latches that occupy approximately 2.5 logic gates per bit, comparators use 2.5 logic gates per bit while tri-state buffer use 0.5 logic gates per bit. The *mtf1* MRU stack size is estimated at approximately 8400 gates. Similarly the *mtf2* size is estimated to be approximately 150 logic gates. Together with the LV predictor (12-bit register + comparator) and the AZLC counter (4 bits) the total complexity of the DMTF(192,4) is less than 24,600 gates.

6. CONCLUSIONS

This paper presents the double move-to-front method for program trace compression that successfully exploits temporal and spatial locality of program streams to achieve compression ratios of two orders of magnitude. Detailed evaluation of its effectiveness on a diverse set of benchmarks shows that the compression ratio for our best performing configuration ranges between 82.7:1 and 29,389:1 (268 on average), that translates into trace port bandwidth of 0.001 to 0.39 bits/instruction (0.12 bits/instruction on average). We have introduced a cost-effective implementation of the proposed program trace compressor. The best performing configuration has an estimated complexity equivalent to 25,000 logic gates, which is a half of the complexity reported for the LZ-based trace compressor [5].

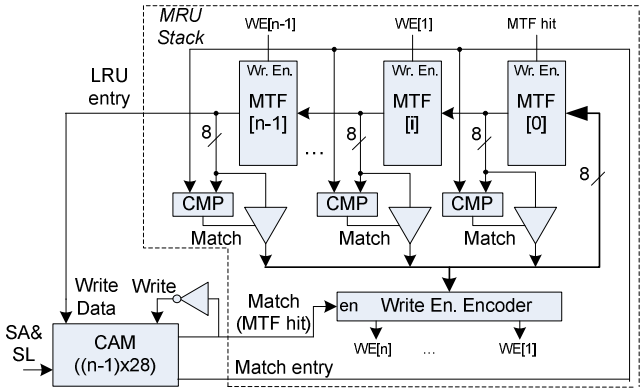

Figure 8. MTF Hardware Implementation.

The proposed trace compressor allows designers to effectively trade complexity and compression ratio, depending on application characteristics and available on-chip area for the trace module. For example, with DMTF(64, 4) we achieve 0.2 bits/instruction on average (ranging from 0.001 to 1) at the cost of 8,200 logic gates. With DMTF(128, 4) we achieve 0.16 bits/instruction on average (ranging from 0.001 to 0.6) at the cost of 16,500 logic gates.

7. REFERENCES

[1] ARM, "Embedded Trace Macrocell Architecture Specification," http://infocenter.arm.com.

[2] Altera, "Nios II Processor Reference Handbook," http://www.altera.com.

[3] Xilinx, "MicroBlaze Processor Reference Guide Embedded Development Kit EDK 10.1i," http://www.xilinx.com.

[4] "Lauterbach GmbH," http://www.lauterbach.com.

[5] C.-F. Kao, et al., "A Hardware Approach to Real-Time Program Trace Compression for Embedded Processors," *IEEE Transactions on Circuits and Systems,* vol. 54, pp. 530 - 543, 2007.

[6] M.-C. Hsieh and C.-T. Huang, "An embedded infrastructure of debug and trace interface for the DSP platform," in *45th ACM Design Automation Conference*, 2008.

[7] M. Milenkovic, et al., "Algorithms and Hardware Structures for Unobtrusive Real-Time Compression of Instruction and Data Address Traces " *Data Compression Conference,* pp. 283-292, 2007

[8] M. R. Guthaus, et al., "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE 4th Workshop on Workload Characterization*, 2001.

[9] T. Austin, et al., "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer,* vol. 35, pp. 59-67, 2002.

[10] B. Jon Louis, et al., "A locally adaptive data compression scheme," *Commun. ACM,* vol. 29, pp. 320-330, 1986.

[11] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital SRC Research Report 1994.

[12] K. Pagiamtzis and A. Sheikholeslami, "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey," *IEEE Journal of Solid-State Circuits,* vol. 41, 2006.