REAL-TIME DETECTION OF FINGER PICKING MUSICAL STRUCTURES

Dale E. Parson

Agere Systems, Allentown, Pennsylvania, USA
dparson@agere.com

ABSTRACT

MIDIME is a software architecture that houses improvisational agents that react to MIDI messages from a finger-picked guitar. They operate in a pipeline whose first stage converts MIDI messages to a map of the state of instrument strings over time, and whose second stage selects rhythmic, modal, chordal, and melodic interpretations from the superposition of interpretations latent in the first stage. These interpretations are nondeterministic, not because of any arbitrary injection of randomness by an algorithm, but because guitar playing is nondeterministic. Variations in timing, tuning, picking intensity, string damping, and accidental or intensional grace notes can affect the selections of this second stage. The selections open to the second stage, as well as the third stage that matches second stage selections to a stored library of composition fragments, reflect the superposition of possible perceptions and interpretations of a piece of music. This paper concentrates on these working analytical stages of MIDIME. It also outlines plans for using the genetic algorithm to develop improvisational agents in the final pipeline stage.

1. MIDIME SOFTWARE PIPELINE

MIDIME is a software architecture that distills the structure of finger-picked string music at several levels of abstraction. It accepts MIDI input from a guitar, analyzes playing to determine low level (finger patterns, meter, accents, tempo, root, scale and chords) and high level (composition fragments) musical intent, and generates accompaniment MIDI streams for one or more synthesizers. This paper is about the architecture and algorithms of MIDIME's fully working analysis stages.

Figure 1 illustrates the MIDIME pipeline. MIDI input through Stage 3 and MIDI output are in full working form. Stage 4 is in a working prototype state. Each stage in Figure 1 is a software thread that analyzes data from the preceding stages and writes its interpretations to its output data table. A data table is a memory resident data structure which its writer updates and which downstream writers read. Each data table is a first-in first-out circular queue of rows owned by its writer. A writer controls its data table by locking the single data row that it is working on until it makes a change that it must expose to downstream readers; the writer then releases the lock, locks the following row, and copies the completed row into the new row, where it repeats the cycle. This is a conventional queuing architecture, with the novelty that reader threads entail the overhead of blocking on a lock only when polling determines that they need access to the row under construction.

The contents of a data table depend on its pipeline stage. A row in a Stage 1 data table contains guitar string state for a given time period. A Stage 2 row contains musical structure data for a given time period. A Stage 3 row contains indices showing where the current performance matches the state of a composition map

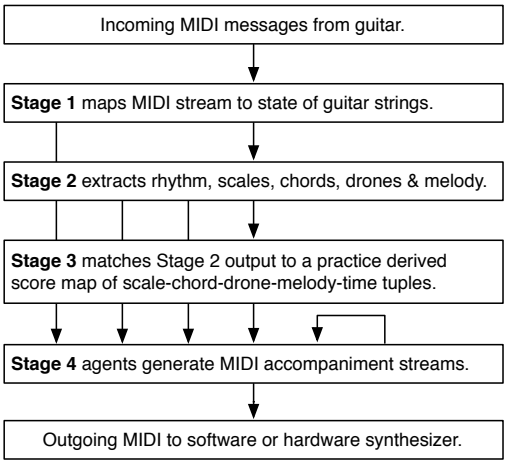

Figure 1: *MIDIME pipeline architecture.*

derived from practice performances. A Stage 4 row contains outgoing MIDI messages. Each writer is alternately a reader that waits for new data from upstream writers, and a writer that emits another row to downstream readers.

Overall, the pipeline of Figure 1 acts as a reactive system that responds to incoming messages within application time constraints. Each writer has the job of waiting for data tailored to its application requirements within the pipeline, transforming the data to meet the requirements of the next downstream writer, and releasing that data. Analysis steps are sometimes complex, but they are always restricted to a tractable single level of abstraction.

2. GUITAR STRINGS OVER TIME

Stage 1 converts a stream of finger-picked MIDI messages to a two dimensional matrix of guitar string state over time, where the most recent row is the current state of the strings, and each column is a string. Writer constructor parameters establish the number of strings, MIDI channel-to-string mappings, and minimum noteon velocity and duration. Stage 1 discards any note with velocity or duration below its minimum threshold as a transient.

The main jobs of the Stage 1 writer are to map MIDI channels-to-string positions, map pitch bend messages to discrete semitones while discarding extraneous pitch bends, record string / note state in a Stage 1 row, and release that row to downstream readers. The guitar synth uses one MIDI channel per string.

Table 1 shows a short trace of the state of a pluck on the E note at the second fret of the fourth string, followed by a slide to the F at the third fret. The left column shows incoming MIDI messages and the right column shows Stage 1 data rows as they appear in the MIDIME debug graphical user interface. A MIDI trace includes buffer row number, message type, channel, velocity for noteon and

magnitude for bend, and arrival time in milliseconds since the start of the process. A Stage 1 trace shows row number, changing string number, with subsequent lines showing the state of all six strings: musical note and octave on the string (e.g., E4), "∼" showing cumulative bend, "ˆ" showing string velocity, "o" showing volume setting, "<"showing note start time in milliseconds, ">" showing duration for notes just terminated, and a hex line for MIDI control signals such as foot pedal messages. The zeroed volume and control lines are elided from most of the Stage 1 trace in Table 1.

0: bend,c=4,b=0x2068	0: string 4, stage1 row 0:
2549	−−− −−− E4 −−− −−− −−−
1: noteon,c=4,p=E4,v=122	∼0 ∼0 ∼8192 ∼0 ∼0 ∼0
2550	ˆ0 ˆ0 ˆ122 ˆ0 ˆ0 ˆ0
2: bend,c=4,b=0x2074	o0 o0 o0 o0 o0 o0
2569	<0 <0 <2550 <0 <0 <0
3: bend,c=4,b=0x2088	0> 0> 0> 0> 0> 0>
2598	0x0 0x0 0x0 0x0 0x0 0x0
4: bend,c=4,b=0x2098	1: string 4, stage1 row 1:
2619	−−− −−− E4 −−− −−− −−−
5: bend,c=4,b=0x20a2	∼0 ∼0 ∼8258 ∼0 ∼0 ∼0
2729	ˆ0 ˆ0 ˆ122 ˆ0 ˆ0 ˆ0
6: bend,c=4,b=0x2092	<0 <0 <2550 <0 <0 <0
2789	0> 0> 329> 0> 0> 0>
7: bend,c=4,b=0x2070	2: string 4, stage1 row 2:
2820	−−− −−− F4 −−− −−− −−−
8: bend,c=4,b=0x2042	∼0 ∼0 ∼8206 ∼0 ∼0 ∼0
2849	ˆ0 ˆ0 ˆ122 ˆ0 ˆ0 ˆ0
9: bend,c=4,b=0x22b8	<0 <0 <2879 <0 <0 <0
2879	0> 0> 0> 0> 0> 0>

Table 1: *MIDI & Stage 1 trace of a semitone slide on string.*

The most substantial job of the Stage 1 writer is filtering the many pitch bend messages emitted by the guitar synth. The Roland GR-33 guitar synth does not send a new noteon message when sliding into a new semitone. Instead, it emits a series of pitch bend messages centered at 8192 (0x2000), with the GR-33 configured to bend down from 8192 to 0 over an octave range, and to bend up from 8192 to 16383 (0x3fff) over an octave range. This configuration gives a semitone bend value of 8192/12 or about 683 (0x2ab). Table 1 shows the slide over the fret that exceeds the 0x2ab threshold in the last MIDI line with a bend of 0x22b8 at time 2879 ms., about 1/3 of a second after the noteon message showing the string pluck. Stage 1 integrates this MIDI stream into three rows that show the E4 pluck, the end of E4 when crossing the fret at 2550+329 = 2879 ms., and the F4 note starting at 2879 ms. Each "∼" bend within a string's Stage 1 row is residual bend left over after computing the fret crossings. As in the Table 1 example, Stage 1 always interprets *slurs* on a sounding string (*i.e.*, hammering onto a string, pulling off, sliding and stretching with a fretting finger) as new string state with zero time elapsed from the previous string state, e.g., E4 start time 2550 + duration 329 = F4 start time 2879 in this example. Fresh *plucks* with a picking finger always show non-zero time elapsed between note states on a string, *i.e.*, last note time + duration $<$ next note time. Distinguishing plucks from slurs is critical for Stage 2 analysis, since plucks distinguish meter and tempo in finger picking patterns.

Table 1 shows a 3-to-1 reduction in the number of MIDI-to-Stage 1 events delivered to downstream readers. The overall trace for this test run of various left hand slurs shows about a 5-to-1 reduction, which is fairly typical for guitar playing and for GR-33 sensitivity and stability. Transitory pitch bends accumulate in the Stage 1 "∼" entries, but only semitone-crossing pitch bends advance a Stage 1 row. There is no reduction in Stage 1 data size; entries are largely redundant with previous rows, in the interest of making the entire state of all strings at a point in time easily accessible to downstream analysis. The typical 5-to-1 reduction in events does greatly reduce the processing demands on downstream analysis, however. The Stage 2 writer activates for as few as one fifth of the incoming MIDI messages, and when it does, the Stage 1 data rows that appear in Figure 2 are structured to ease analysis of finger picking patterns and string-oriented note analysis.

3. MUSICAL STRUCTURES OVER TIME

3.1. Finger Patterns, Meter and Tempo

When a Stage 2 writer receives a new Stage 1 row, its first step is to analyze rhythm. It starts with rhythm because subsequent analyses of scale, chord, drones and melody depend on performer-established time frames. Scales and drones are long term entities. Chords are usually short term entities that may cycle through long term patterns. Establishing long and short term temporal boundaries requires establishing a temporal reference.

Unlike many computer music systems, MIDIME does not impose an external, metronome-like temporal grid on incoming or outgoing notes. Finger picking is intrinsically a repetitive, rhythmic activity that establishes its own temporal boundaries and cycles. The picking hand plucks the strings in series of repeating rhythmic patterns while the other hand frets, slides, hammers, pulls and stretches the strings in order to select and alter pitch. The timing and force of finger plucks in a pattern show considerable surface variation, but behind this variation is an intent to create a cyclic pattern. The problem for Stage 2 is to find patterns in data representing musical intent, in rhythm or tone, while dealing with seemingly random fluctuations in surface features. There may be more than one possible rhythmic or tonal interpretation of incoming performance data; that is OK, because it gives Stage 4 improvisers the opportunity to go into unexpected directions that are empirically related to the human performance. The approach of Stage 2 is to locate regularities that are abstractions of the incoming notes. This temporary gridding is counteracted by Stage 4 writers who map the Stage 2 abstracted data back onto the real time and note space being played.

The example rhythm analysis of this section uses the finger picking pattern of Figure 2. The thumb (T) picks the first of 8 eighth notes in a 4/4 measure, followed by eighth notes on the index finger (I) on an intermediate string and the middle finger (M) on the first string. This pattern repeats, then the thumb strikes again on the seventh eighth note, and on the eighth there is a rest while the thumb prepares to strike again. This repetition of the thumb is not normally found in bluegrass, which often uses the three fingers in basic triplets. Thumb repetition slows the tempo and places accents on the first, fourth, and seventh eighth notes. Sometimes the index or middle finger dampens its string instead of plucking.

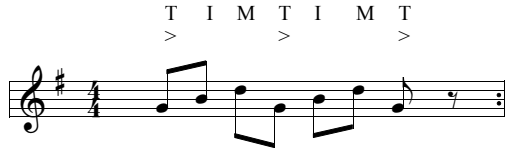

Figure 2: *4/4 Finger pattern with accents at 1, 2.5, and 4.*

The algorithm to extract meter and tempo from Stage 1 data works in the following steps.

1. Collect pointers to Stage 1 string plucks into a 24 element array; collect records of the millisecond intervals since the previous respective plucks into a corresponding array of integers. An array size of 24 allows multiple repetitions of all possible three-finger patterns. There may be remainder plucks at the beginning or end of the array when the picking pattern does not divide 24 plucks evenly, or when picking goes through a pattern or tempo transition. Slurs do not contribute because only plucks represent right hand picking patterns. When the array fills, the oldest pluck is discarded at the start of event processing.

2. Build a temporary array of pointers to the elements of the above interval array, and sort this temporary array by the interval time in ascending order, giving access to the shortest interval first.

3. Traverse this sorted array and build a third array of bins. The smallest interval goes into the first bin, along with all succeeding intervals that are not more than 20% longer than the least interval in the bin. This 20% threshold was originally 16% in anticipation that a given unit of time could be subdivided into either 2 or 3 or (2×3) for conventional note subdivisions such as quarter-to-eighth note or quarter-to-triplet subdivisions; $1/(2 \times 3) = 1/6 = 16.7\%$. This attempt was too tight for typical playing; relaxing it to 20% made matching more consistent. As soon as a pluck interval exceeds 20%, it becomes the least value in a new bin; longer intervals go into that bin until exceeding its 20% threshold, creating a new bin, and the process continues until all 24 intervals are placed into bins.

4. Each bin is averaged by dividing its sum by the number of intervals in the bin. This number gives average picking time for that bin, with the bin representing some part of a pattern. The arithmetic division for averaging, like all division in MIDIME, is scaled integer division rather than slower floating point division.

5. Take the entire interval time in the 24 element array of plucks as a conceptual "measure" — this time is entirely empirical and is not tied to any rigid framework — and begin subdividing it into grid boundaries. First divide this total time into half, giving three snap points, one at the start (0 time), one at the end (total time), and one halfway between. For each bin from step 4, build a new bin that moves the value in the step 4 bin to its closest snap point, if and only if this snapping step does not move the bin average more than 20%. If any snap would exceed 20% in either direction, try a new snap by dividing the total time by the next step in the series (2, 3, 4, 6, 8, 12, ...). This series again represents splitting time repeatedly into halves and triplets. Eventually a subdivision of total time is found that allows snapping each average bin interval by less than 20%. In the worst case the total time would be divided down to 1 millisecond snap units, giving no snapping, but any regularity in the performer's finger picking patterns avoids this degenerate state. Create a snapped interval for each bin when the process converges.

6. It is now possible to divide each snapped interval value by the snap unit to come up a small integer for each bin, and small integer ratios across bins that display repetition representing finger picking patterns. This step is implicit. The snapped bins of step 5 are sufficient for fast, machine level block memory comparisons.

7. Look for repeating patterns in the values of the bins of step 5. Find the shortest repeating pattern of snapped interval values that covers the largest number of elements in the 24 element interval array. For example, repeating pattern Z-X-Y-X-Y Z-X-Y-X-Y would not subdivide further because these two five-element patterns organize ten total intervals, while an X-Y X-Y pattern would organize only the four elements at the end of the array. A pattern such as X-Y-X-Y-X-Y, in contrast, would factor into three adjacent X-Y patterns covering six contiguous intervals.

The algorithm proceeds by starting with a pattern length of one half the 24 elements, using the C library function memcmp to compare adjacent subsequences of snapped intervals. After comparing all adjacent pairs of sequences at a given length, the algorithm reduces the proposed pattern length by 1, and again compares adjacent subsequences of snapped intervals. It proceeds down to a proposed pattern length of 1, always storing the smallest successful pattern length that matches the greatest number of intervals. If it finds no matches, then it treats the entire array of intervals as one, non-repeating pattern. This condition usually indicates that the picking pattern or tempo is changing.

Table 2 makes these steps concrete with an example using real data. *Raw time* shows measured time intervals since previous plucks in milliseconds. The smallest value in each 20% bin is highlighted in the raw time row (*i.e.*, 125, 156, 189 and 333). *Bin average time* averages raw values in each bin, yielding average bin intervals 143, 175, 195 and 341. Splitting the total time of 5.125 seconds for this sample yields an interval snap of 160, a rounding of $(5125/32) = (5125/2^5)$, or 5 halvings of the total concrete "measure." This is the coarsest splitting of the total time that snaps each average bin time within 20% of its value. Pattern matching then determines that a repeating pattern seven intervals in length covers the last 14 intervals in this example; the pattern is 1-1-1-1-1-1-2 snap units; the final 7 are highlighted. These are the eighth notes of Figure 2 with the unplucked rest residing halfway through the final 2 interval; the first eighth note of Figure 2 is the last pluck of the detected pattern. Rhythm analysis also normalizes pluck MIDI velocities as scaled multiples of the average pluck velocity for each slot in a detected pattern, giving a velocity pattern of 1-1-3-1-0-4-3 for this table's data. The string numbers are not used in analysis, but for this data they are 2-1-3-2-1-3-3. Finally, the fingers used to pluck this data are I-M-T-I-M-T-T. Note the two-eighth-note relative timing of the adjacent thumbs, and the scaled velocities of 3 and 4 for the thumb. Stage 4 processing can use the final seven slots of this table to determine tempo and accents and to schedule output improvisation timing and velocity.

A player will occasionally dampen a string in this pattern so that no pluck will be detected, sometimes intentionally, and will sometimes hit an extra string by accident. These variations add intervals that match combined intervals or subintervals of the pattern. Rhythm analysis determines if there are any trailing sequences of intervals after the final pattern sequence whose total snapped times add up to the pattern time. If it finds them, it marks them as "tail patterns" that include missing or extra plucks. There are none in Table 2, although there are 6 plucks that add 8 snap units highlighted near the top of the right column, preceding the first occurrence of the pattern. The index finger dampened its string for the out-of-place 2. Rhythm analysis treats trailing sequences that sum to pattern time as additional cases of the pattern for purposes of scheduling output. Downstream improvisation has access to snapped and raw times in planning its output note scheduling.

Irregularities in playing and transitions in pattern and tempo cause periods of loss of pattern detection. There are two ways for Stage 4 accompaniment agents to deal with periods of desynchronization. The first is to treat such periods as *rubato*, *i.e.*, instrinsically desynchronized periods of playing. In this approach a Stage 4 agent takes the entire 24 interval pluck time as the mea-

raw time in milliseconds	bin average time	snapped time	snaps
186	175	160	1
358	341	320	2
153	143	160	1
333	341	320	2
201	195	**160**	**1**
148	143	**160**	**1**
194	195	**160**	**1**
348	341	**320**	**2**
144	143	**160**	**1**
337	341	**320**	**2**
203	195	160	1
146	143	160	1
178	175	160	1
179	175	160	1
193	195	160	1
125	143	160	1
339	341	320	2
191	195	**160**	**1**
156	175	**160**	**1**
177	175	**160**	**1**
173	175	**160**	**1**
189	195	**160**	**1**
141	143	**160**	**1**
333	341	**320**	**2**

Table 2: *Temporal intervals for a seven-pluck finger pattern.*

sure, subdividing it for note generation according to the agent's own algorithm. The guitar player's and the agent's note timing weave in and out of overlap in temporal consonance / dissonance transitions. The second, more conservative approach is to program a Stage 4 agent to save a copy of the interval ratios and tempo of the most recent repeating Stage 2 pattern. This agent generates notes that adhere to this pattern and tempo. An example is a bass playing agent emitting a rhythmically stable bass line initialized from repetitive Stage 2 output. The guitar player can follow this emitted bass line, or the guitar player can force transition to a new meter or tempo by playing consistently for rhythmic pattern detection. The guitar player and the Stage 4 agent can pass control of rhythm back and forth as part of improvisation. Prototype MIDIME agents have used both approaches successfully.

Despite having $O(n^3)$ time complexity on the interval array, rhythm analysis and in fact all processing through Stage 3 has a total latency in reacting to a MIDI message that is under the 1 millisecond resolution of the operating system's time function. The combination of machine-level bit and logical operations and the use of scaled integer arithmetic instead of floating point, along with a fast processor and small bounds on the size of data being matched, help to keep pattern matching fast. Stage 1 events have about a 2-to-1 ratio to the number of Stage 2 events because most Stage 2 events are driven by plucks; Stage 1 slurs and note off events contribute tangentially to Stage 2 output.

3.2. Scales, Drones, Chords and Melody

Both scale and chord matching collect recent notes into *12-bit bit vectors*. A vector comprises the bottom 12 bits of a 16-bit C++ integer, with bit position 2^0 representing note C, 2^1 note `C#`, ..., 2^{11} note B. A bit vector discards octave and repeated note information. Scale matching uses an array of 12 data structures to track pluck times and to sum the number of pluck appearances of each note C .. B in the last four measures, counting a given note only once per measure, where a measure is the finger pattern time just discussed. Thus a note C .. B can be counted 0 to 4 times in four measures.

After updating pluck counts and retiring any plucks older than four measures, scale analysis starts by using only the notes with the highest number of plucks. For example, if some notes appeared 4 times, only those notes would contribute their bits into a 12-bit scale vector. Scale analysis uses pattern matching described next to match this 12-bit vector to a scale, recording the scale if it matches. It then tries again with all notes with counts of 4 or 3, then again with 4 or 3 or 2, and finally with 4 or 3 or 2 or 1. The intent in starting with 4 is to discard transient, chromatic notes that appear infrequently during initial matching. If this approach discards too many notes to find an effective match, the later attempts with more notes may succeed. All four attempts are always made, but a later scale match replaces an earlier match of fewer notes only if the later match is a better match.

Scale analysis looks for a match in two loops. The first, *biased loop* iterates through optional 16-bit *scale configuration parameters* supplied by the user. If the user knows scales in advance, he or she can configure Stage 2 of the pipeline to try these scales. A 16-bit scale parameter consists of a 12-bit note pattern as before, plus a 4-bit number 0 .. 11 representing the root of the scale, C .. B, in the top 4 bits of the 16-bit vector. The biased loop works by bitwise ORing these configuration parameter bits into the played, empirical bits extracted from the performance as described by the last paragraph, and attempting to match that vector. This is a top down approach.

Scale configuration parameters are optional, and even when they are available, scale matching also uses an *empirical loop* to try to match the empirical bits. This matching loops through an optional set of *tonic configuration parameters*, which are some subset of the values 0 .. 11 representing a tonic of C .. B. If these optional parameters are missing, MIDIME simply tries all possible tonics 0 through 11 in combination with the empirical bits, by shifting the tonic bits into the top 4 bit positions of a 16-bit vector. Scale analysis maps a given 16-bit vector to a possible scale by using the 16-bit vector as an integer index into a 53248-entry table of scales (12 tonics + a key for unknown tonic = 13 keys $\times 2^{12}$ combinations of 12 notes = 53248). Here is a possible match for the note pattern 0x0a5 in the key of D, *i.e.*, notes C-D-F-G

```
const uint16_t MidimeModeTable[53248] = {//lots of entries
      0x2a5, // 0x20a5 --> D minor_pentatonic, distance 1
```

The comment after "//" shows that 0x20a5, C-D-F-G rooted in D, maps to the value 0x2a5, notes C-D-F-G-A, with a distance of 1. This is the D minor pentatonic scale. The distance shows that the actual data misses the table's entry by 1 bit. The total difference in number of bits is the *Hamming distance*. It measures how good the match is, and this table stores mappings only with distances of 2 or less. Invalid mappings appear as 0 in the table, *i.e.*, no notes on.

This table is generated by another program and compiled into C++ MIDIME. Consulting the table takes small constant time to determine a matching scale. Hamming distance is determined by exclusive-ORing the biased or empirical note vector played with the scale vector proposed by the table, giving an integer with 1 in each bit position where the real notes differ from the table's notes. There would be 1 bit for the missing A note in this example. Scale analysis uses this Hamming difference vector as an index into a compiled Hamming weight table that returns the number of bits set to 1 for that entry. The Hamming weight tells how far the match is

from perfect, and scale analysis stores the best match it has seen so far. At the end of its two loops, scale analysis uses the best match from biased and empirical analysis, using empirical to break a tie. Scale analysis also stores the tonic associated with the best match.

In the absence of optional scale configuration parameters, scale matching tends to be conservative, for example preferring major or minor pentatonic scales to more complex scales by using only notes with high pluck counts in the last four measures. Biased matching, in contrast, tends to prefer configuration parameter scales when they fit with real note data. This result works well for improvisation. In the absence of an anticipated scale, downstream improvisers will have conservative reactions in the absence of concrete data; in cases where the user supplies anticipated scales, the improvisers will use them as long as they fit with the real notes.

Drone matching looks to see an open string plucked repeatedly, without pitch change, over at least 4 finger pattern measures. If a string's pitch changes in these measures, it is dropped from the 12-bit drone vector, and if its pitch drops below previously recorded values, that new pitch becomes the baseline open string pitch.

Chord analysis considers only plucked notes currently sounding on the strings. It maintains a note for each sounding string, discarding low velocity notes as possible mistakes, combining the others into a 12-bit vector, then consulting a chord mapping table and computing Hamming distance similar to scale matching. Chord matching uses only empirical note data. It does not interact with long-term scale matching in order to allow for playing "outside" a scale's harmonic structure, a practice common in jazz. When these data are sparse, chord matching makes conservative selections such as simple triads.

Finally, melody matching use the most recent pluck that is greater than the average velocity within its measure, as determined by meter velocity calculations. This matching aligns well with the practice of using the thumb for melody when finger picking.

4. COMPOSITION MATCHING AND ACCOMPANIMENT

Construction of Stage 3 of the pipeline is recently completed. It uses traces of Stage 2 scale-chord-drone-melody state that a user builds by saving and merging performances of composition fragments. Stage 3 looks for saved scale-chord-drone sequences that match current Stage 2 data within a parameterized Hamming distance limit, similar to Stage 2 Hamming-based matching, by comparing performance to stored fragments. Stage 3 identifies known composition fragments in real time to allow Stage 4 improvisers to make long term, complex improvisation plans.

Stage 4 players are in a prototype state. Agents have chromosomes consisting of long and short term genes of four types: rhythm, melody, harmony and timbre. A gene is a unit of typed code that contributes to a long or short term plan for that gene type. A gene's plan is a reaction to something played, or to a Stage 3 map of where playing appears to be heading, or to another Stage 4 player's performance.

5. RELATED WORK AND CONCLUSION

Roads gives an outline of the tasks required for rhythm analysis, along with a substantial list of references in his chapter on pitch and rhythm recognition [1]. Hamanaka, et. al., have designed a system that uses hidden Markov models to eliminate surface deviations while analyzing note onset times [2], and that uses the generative theory of tonal music to infer hierarchical metrical structure from MIDI performance data [3]. Their techniques comprise statistical analyses of surface variations in meter in a fixed-tempo, fixed-duration set of performances.

Papers on determining edit distance with insertions and deletions in musical score retrieval, a non-real-time generalization of the bit vector Hamming distances used for MIDIME scales and chords, can be found in [4, 5]. Toussaint shows application of Hamming distance to rhythm analysis [6].

MIDIME differs from meter analysis systems cited above in extracting cyclic rhythmic information that is intrinsic in repeated patterns used by finger picking guitarists. By using the most recent 24 real-time string plucks for temporal background, and by looking for patterns only in the performance data as opposed to matching string plucks to a library of known finger patterns, MIDIME can adapt to changes in tempo, meter and duration and to new finger patterns in real time.

Thom's system uses an off-line learning algorithm to create a probabilistic model applied to rewrite four-measure melodies at performance time [7]. Biles uses the genetic algorithm in creating accompaniment melodies [8]. Both frameworks require scripted chord sequences and a tempos. The goals of MIDIME are different from these systems, both in MIDIME's analysis of all aspects of empirical musical data in preference to scripts, and in creating harmonic, rhythmic and timbral behavior, in addition to the melodic orientation of these systems. The primary goal of MIDIME is exploration of new music within a framework of multi-level improvisation, rather than limited improvisation within fixed musical structures found in other performance analysis and improvisational systems.

6. REFERENCES

[1] C. Roads, *The Computer Music Tutorial*. MIT Press, 1996.

[2] M. Hamanaka, M. Goto, H. Asoh, and N. Otsu, "A learning-based quantization: Unsupervised estimation of the model parameters," in *Proc. Int. Comp. Music Conf. (ICMC'03),* Singapore, 2003, pp. 369–372.

[3] M. Hamanaka, K. Hirata, and S. Tojo, "Automatic generation of metrical structure based on GTTM," in *Proc. Int. Comp. Music Conf. (ICMC'05),* Barcelona, Spain, 2005, pp. 53–56.

[4] K. Lemstrom and S. Perttu, "SEMEX – an efficient music retrieval prototype," in *Proc. Int. Symp. Music Information Retrieval (ISMIR'00),* Plymouth, Massachusetts, USA, 2000, [Online] http://ismir2000.ismir.net/papers/lemstrom_paper.pdf.

[5] A. Pienimaki and K. Lemstrom, "Clustering symbolic music using paradigmatic and surface level analyses," in *Proc. Int. Conf. Music Information Retrieval (ISMIR'04),* Barcelona, Spain, 2004, pp. 175–178.

[6] G. Toussaint, "A comparison of rhythmic similarity measures," in *Proc. Int. Conf. Music Information Retrieval (ISMIR'04),* Barcelona, Spain, 2004, pp. 134–137.

[7] B. Thom, "BoB: An improvisational music companion," in *Proc. Fourth Int. Conf. Autonomous Agents*, Barcelona, Spain, 2000, pp. 309–316.

[8] J. A. Biles, "GenJam," [Online] http://www.it.rit.edu/~jab/ including numerous publications on GenJam and evolutionary music.