

**Fast Rule Matching for  
Learning Classifier Systems  
via Vector Instructions**

**Xavier Llorà and Kumara Sastry  
IlliGAL Report No. 2006001  
January, 2006**

Illinois Genetic Algorithms Laboratory  
University of Illinois at Urbana-Champaign  
117 Transportation Building  
104 S. Mathews Avenue Urbana, IL 61801  
Office: (217) 333-2346  
Fax: (217) 244-5705

# Fast Rule Matching for Learning Classifier Systems via Vector Instructions

**Xavier Llorà**

Illinois Genetic Algorithms Laboratory  
National Center for Supercomputing Applications  
University of Illinois at Urbana-Champaign  
1205 W. Clark Street, Urbana, IL 61801, USA  
xllora@illigal.ge.uiuc.edu

**Kumara Sastry**

Illinois Genetic Algorithms Laboratory  
Department of General Engineering  
University of Illinois at Urbana-Champaign  
104 S. Mathews Avenue, Urbana, IL 61801, USA  
kumara@illigal.ge.uiuc.edu

## Abstract

Over the last ten years XCS has become a *de facto* standard for Michigan-style learning classifier systems (LCS). Since the initial CS-1 work conceived by Holland, classifiers (rules) have widely used a ternary condition alphabet  $\{0,1,\#\}$  for binary input problems. Most of the freely available implementations of this ternary alphabet in XCS rely on character-based encodings—easy to implement, not memory efficient, and expensive to compute. Profiling of freely available XCS implementations shows that most of their execution time is spent determining whether a rule is match or not, posing a serious thread to XCS scalability. In the last decade, multimedia and scientific applications have pushed CPU manufactures to include native support for vector instruction sets. This paper presents how to implement efficient condition encoding and fast rule matching strategies using vector instructions. The paper elaborates on AltiVec (PowerPC G4, G5) and SSE2 (Intel P4/Xeon and AMD Opteron) instruction sets producing speedups of XCS matching process beyond ninety times. Moreover, such a vectorized matching code will allow to easily scale beyond tens of thousands of conditions in a reasonable time. The proposed fast matching scheme also fits in any other LCS other than XCS.

## 1 Introduction

Since the publication of the work by Wilson in 1995, XCS (Wilson, 1995) has been widely adopted by the mainstream of researchers working on the so-call Michigan-style learning classifier systems. XCS builds on the origination work by Holland (Holland, 1975; Holland & Reitman, 1978) and his first learning classifier system proposed (CS-1), later revised and improved (SCS) by Goldberg (Goldberg, 1989). XCS, likewise CS-1 and SCS, uses a ternary alphabet  $\{0,1,\#\}$  to describe conditions given a binary input. Most of the freely available implementations of XCS, such as `xcslib`<sup>1</sup> and `XCS-C`<sup>2</sup>, rely on character-based encoding of the conditions described using the ternary alphabet. This approach to condition encoding is easy to implement, but it is not memory efficient (75% of the character allocated memory is not used), and it becomes expensive to compute when we increase the problem size (number of conditions/attributes).

The character-based encoding also poses a thread on the scalability of XCS. When scaling beyond thousands, or tens of thousand attributes, XCS will greatly suffer from memory wasting and

---

<sup>1</sup>`xcslib` maintained by Pier Luca Lanzi and can be downloaded at <http://xcslib.sourceforge.net/>.

<sup>2</sup>`XCS-Cis` maintained by Martin Butz and can be downloaded at <http://www-illigal.ge.uiuc.edu/sourcecd.html>.

inefficient rule (classifier) matching. Profiling the freely available XCS implementations it becomes clear that more than half of the time `xcslib` and `XCS-C` are spending is in the rule matching processes. Hence, one of the goals of this paper is to reduce the memory footprint of the rule encoding schemes used in XCS. Reducing the memory footprint will also help the matching process thanks to a better usage of the underlying processor caches. Moreover, the proposed encoding will also be intended to allow fast matching strategies using widely available hardware instructions. Recently, multimedia and scientific applications have pushed CPU manufactures to include again in their processors support for vector instruction sets. With a proper rule encoding, the usage of vector instructions for parallelizing the matching process can greatly reduce the time spend during the matching process.

This paper presents how to implement efficient condition encoding and fast rule matching strategies using vector instructions. The paper elaborates on two widely spread vector instruction sets: (1) AltiVec (available on PowerPC G4 and G5 processors), and (2) SSE2 (available on Intel P4/Xeon and AMD Opteron processors). Experimental results show the advantages of using vector instruction sets, producing speedups of XCS matching process beyond ninety times when compared to the traditional character-based encoding. Moreover, such a vectorized matching code will allow to easily scale beyond tens of thousands of conditions in a reasonable time.

The rest of this paper is structured as follows. Section 2 briefly reviews the freely available implementations of XCS focusing on the execution profiles of `xcslib` and `XCS-C`. Then, section 3 explores how to reduce the memory footprint of XCS rules. Given the new rule representation, section 4 elaborates on matching implementations that take advantage of vector instruction sets. Time measures taken using the proposed fast vector-based matching implementations show the usefulness of the proposed approach, as shown in section 5. Finally, section 6 discusses some conclusions and further work.

## 2 XCS implementations

The work presented in this section uses two of the freely available XCS implementations: (1) `xcslib` by Pier Luca Lanzi and (2) `XCS-C` by Martin Butz. A detailed description of both implementation is beyond the scope of this paper and can be found elsewhere (Butz & Wilson, 2000; Lanzi, 2005). For each XCS implementations, the rule-encoding scheme is briefly reviewed. Then, statistical execution profiles are obtained and analyzed using four different multiplexer problems, providing interesting insides about the execution behavior of both XCS implementations.

### 2.1 The widely used character-based encoding

One of the most widely spread XCS rule implementation encodes each condition as a character. Each condition, hence, is represented by a '0', '1', or '#', and the matching of a rule consists on checking whether the character is the same or the rule contains the *don't care* character. Figure 1 presents the code for rule matching using conditions based on character-encoding, and the different implementations provided by `xcslib` and `XCS-C`. Both implementations exploit the simple optimization of stopping the matching process once a condition is not matched. Regardless of the optimization, the matching of a rule behaves as  $\mathcal{O}(\ell_c)$ —being  $\ell_c$  the number of conditions.

### 2.2 Rule sets as lists

`xcslib` and `XCS-C` store the rules to match in list-like data structures. Thus, both implementations also require linear times  $\mathcal{O}(\ell_r)$  to determine the match set [M] once a new instance is provided.

Figure 1: Default character-based rule implementation and matching. The commented code is a simple improvement to the linear exploration of conditions. The figure also presents the matching code of the freely available `xcslib` and `XCS-C` implementations of XCS.

---

```
#define RULE char*

int isRuleMatched ( RULE r, INSTANCE i ) {
    int iMRCL,iFlag;

    for ( iMRCL=0, iFlag=1 ;
          iMRCL<NUM_CONDITIONS /*&& iFlag*/ ;
          iMRCL++ )
        if ( i[iMRCL]!=r[iMRCL] && r[iMRCL]!='#' )
            iFlag=0;

    return iFlag;
}
```

---

XCSlib implementation

```
bool
ternary_condition::match(const binary_state& sens)
{
    string::size_type bit;
    string input;
    bool result;

    input = sens.string_value();
    assert(input.size()==bitstring.size());

    bit = 0;
    result = true;

    while ( (result) && (bit<bitstring.size()) ) {
        result = ( (bitstring[bit]=='#') ||
                  (bitstring[bit]==input[bit]) );
        bit++;
    }

    return result;
}
```

---

XCS-C implementation

```
int match(char *m, char *c)
{
    for( ; *c!='\0'&&*m!='\0' &&
          (*m==DONT_CARE||*c==DONT_CARE||*c==*m) ;
          m++,c++ );
    if (*m=='\0' || *c=='\0')
        return 1;
    else
        return 0;
}
```

---

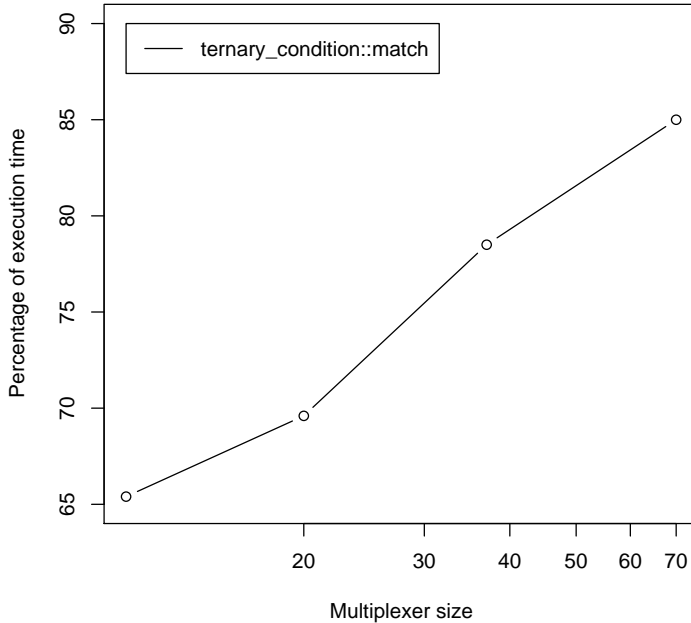


Figure 2: Percentage of execution time spent by XCSlib in the rule matching process (`ternary_condition::match`).

Hence, the overall time required for the matching process grows as  $\mathcal{O}(\ell_r \cdot \ell_c)$ . In the same manner, the memory footprint required to store the population of rules also grows as  $\mathcal{O}(\ell_r \cdot \ell_c)$ . Recent studies have proposed to use index structures to speedup the matching process (Giraldez, Aguilar-Ruiz, & Riquelme, 2005). Tree-based indexing may reduce the required time for matching and instance to  $\mathcal{O}(\ell_r \cdot \log(\ell_c))$ . Such indexing approach can reduce the amount of time required by the matching process. However, such approaches are better fitted when relatively small training data sets are available because of the indexing approach greatly increase the memory footprint required. For instance, indexing a given XCS rule set will require a memory footprint that would grow as  $\mathcal{O}(2^{\ell_c})$ , making the approach inoperative when scaling beyond a few teens of conditions and add extra maintenance cost of such an index. For this reason, the work presented in this paper maintains the basic list based approach.

### 2.3 Statistical execution profiles

The matching process, despite of its importance, is still one of the components of XCS. Although the intuition may point it as a main bottleneck in the quest for XCS scalability, supporting facts are still needed. Obtaining statistical execution profiles for `xcslib` and `XCS-C` provides the required facts to prove or refute such an intuition. Table 1 presents the summaries of the statistical execution profiles for `xcslib` and `XCS-C` when solving the 11-input, 20-input, 37-input, and 70-input multiplexer problems. Both implementations were compiled using `gcc` compiler version 4.0.0 using the `-O3` flag on. The profiling environment used was a PowerBook G4 at 1.5GHz and 1Gb of ram running Mac OS X 10.4.4. The statistical execution profiles were obtained and analyzed using Shark v.4.2.0—see table 1 for further detail.

Table 1: Statistical execution profiles for `xcslib` and `XCS-C` using the XCS configuration parameters provided by the XCSlib distribution. Each implementation was compiled and ran to solve the 11-input, 20-input, 37-input, and 70-input multiplexer. Both implementations were compiled using `gcc` version 4.0<sup>a</sup> with the `-O3` flag on. The execution environment was a PowerBook G4 at 1.5GHz and 1Gb of RAM running Mac OS X 10.4.4<sup>b</sup>. The statistical execution profiles were obtained and analyzed using Shark v.4.2.0<sup>c</sup>.

<i>XCSlib version 0.34</i>			
<i>%</i>	<i>function (11-input multiplexer)</i>	<i>%</i>	<i>function (20-input multiplexer)</i>
65.4%	<code>ternary_condition::match</code>	69.6%	<code>ternary_condition::match</code>
8.4%	<code>xcs_classifier_system::select_delete_rw</code>	10.2%	<code>xcs_classifier_system::select_delete_rw</code>
7.5%	<code>binary_state::string_value</code>	7.5%	<code>binary_state::string_value</code>
5.7%	<code>experiment_mgr::perform_experiments</code>	3.1%	<code>xcs_classifier_system::match</code>
3.8%	<code>xcs_classifier_system::match</code>	2.7%	<code>experiment_mgr::perform_experiments</code>
0.9%	<code>xcs_random::dice</code>	1.0%	<code>xcs_classifier_system::update_fitness</code>
0.9%	<code>multiplexer_env::begin_problem</code>	0.7%	<code>action_base&lt;boolean_action&gt;::operator==</code>
0.9%	<code>xcs_classifier_system::update_fitness</code>	0.5%	<code>xcs_random::dice</code>
<i>time</i>	<i>function (37-input multiplexer)</i>	<i>%</i>	<i>function (70-input multiplexer)</i>
78.5%	<code>ternary_condition::match</code>	85.0%	<code>ternary_condition::match</code>
6.5%	<code>xcs_classifier_system::select_delete_rw</code>	6.3%	<code>binary_state::string_value</code>
6.3%	<code>binary_state::string_value</code>	3.1%	<code>xcs_classifier_system::match</code>
3.2%	<code>xcs_classifier_system::match</code>	1.1%	<code>experiment_mgr::perform_experiments</code>
1.4%	<code>experiment_mgr::perform_experiments</code>	0.8%	<code>ternary_condition::~ternary_condition</code>
0.6%	<code>xcs_classifier::match</code>	0.7%	<code>ternary_condition::cover</code>
0.6%	<code>ternary_condition::~ternary_condition</code>	0.6%	<code>xcs_classifier::match</code>
0.4%	<code>ternary_condition::cover</code>	0.5%	<code>ternary_condition::string_value</code>
<i>XCS-C version 1.1</i>			
<i>%</i>	<i>function (11-input multiplexer)</i>	<i>%</i>	<i>function (20-input multiplexer)</i>
36.2%	<code>deleteStochClassifier</code>	31.9%	<code>deleteStochClassifier</code>
31.3%	<code>getMatchSet</code>	26.6%	<code>getMatchSet</code>
11.1%	<code>addClassifierToSet</code>	9.4%	<code>freeSet</code>
6.2%	<code>freeSet</code>	8.1%	<code>addClassifierToSet</code>
2.7%	<code>updateFitness</code>	3.5%	<code>resetState</code>
1.7%	<code>discoveryComponent</code>	3.2%	<code>updateFitness</code>
1.5%	<code>getPredictionArray</code>	2.3%	<code>drand</code>
1.3%	<code>getActionSet</code>	2.1%	<code>getPredictionArray</code>
<i>time</i>	<i>function (37-input multiplexer)</i>	<i>%</i>	<i>function (70-input multiplexer)</i>
32.2%	<code>getMatchSet</code>	40.0%	<code>getMatchSet</code>
29.0%	<code>deleteStochClassifier</code>	34.0%	<code>deleteStochClassifier</code>
9.2%	<code>freeSet</code>	7.5%	<code>freeSet</code>
8.1%	<code>addClassifierToSet</code>	5.1%	<code>addClassifierToSet</code>
3.4%	<code>updateFitness</code>	3.3%	<code>discoveryComponent</code>
2.3%	<code>getPredictionArray</code>	3.1%	<code>updateFitness</code>
2.1%	<code>getActionSet</code>	2.7%	<code>getPredictionArray</code>
1.8%	<code>resetState</code>	1.7%	<code>getActionSet</code>

<sup>a</sup> <http://gcc.gnu.org/>.

<sup>b</sup> <http://www.apple.com/macosx/>.

<sup>c</sup> <http://developer.apple.com/tools/sharkoptimize.html>.

The statistic profile of `xcslib` presents a clear trend; `ternary_condition::match` will eventually take over most of the execution time as the problem size (number of conditions) is increased. Figure 2 shows the increase of the percentage of time spent in the matching process; `xcslib` also presents a consistent behavior as the problem size is increased. Another interesting property of `xcslib` profile is that, when properly implemented, the a list-based storage of the rule set does not pose a thread to XCS scalability, as the amount of time spend on `xcs_classifier_system::match` shows.

The behavior of XCS-C is not as clear and consistent as `xcslib`. Besides the matching procedure (`getMatchSet`), XCS-C has a secondary element that may govern its scalability: `deleteStochClassifier`. When the problem size grows beyond a certain threshold, the matching process eventually governs the execution time—as table 1 shows. However, it is quite unexpected that XCS-C spend roughly the same amount of time stochastically deleting rules (`deleteStochClassifier`). For this reason, the rest of the work presented in this paper may benefit XCS-C, but not as much as `xcslib` until `deleteStochClassifier` is revisited.

## 2.4 How big is big?

XCS has successfully solved large multiplexer problem up to 70-input one (Butz, Kovacs, Lanzi, & Wilson, 2004). However, how far can XCS be pushed? As mentioned above, the matching process needs to be improved to combine compact rule encoding with a small memory footprint and fast matching strategies. The faster XCS computes the match set [M] can be created, the larger the number of instances can be processed and, hence, larger the problems that can be brought from tractability to practicality. The rest of this paper focuses on this particular point, focusing on exploring large problem sizes—thousands and tens of thousands of attributes. If the time spent in the matching process can be cut to reasonable amount, larger problems may be now at reach.

## 3 Compact rules via bit encoding

This section focuses on reducing the memory footprint of the population of rules used by XCS. A bit-based encoding will provide the compaction desired for the ternary encoding. The encoding proposed will also be a first step toward the parallel matching using vector instruction sets proposed in the next section.

### 3.1 The memory allocation

The ternary representation only requires two bits to encode the three different symbols. However, the character-based encoding is using eight bits, wasting a 75% of the memory provided. Thus, the memory required to store a the population of rules is

$$\mathcal{M}([P]) = \ell_r \cdot \ell_c \cdot \text{sizeof}(\text{character}) \quad (1)$$

Thus, for a set of 5,000 rules given a 1,000 conditions problem, the amount of memory required would be 4,882Kb. Wasting the 75% of the memory also has another byproduct. Current processors have between 1 and 2 Mb of level two cache memory. Wasting a 75% of the memory available implies that for each instance, the matching process will eventually require to heavily access main memory to retrieve all the rules due to the need to explore all the rules. Thus, the benefits of caching will be minimized and the latency of fetching rule conditions will increase because of the inefficient encoding that requires repeated accesses to main memory.

### 3.2 Compact rule encoding

As mentioned above only two bits are necessary for encoding a ternary-based condition. The encoding will also need to provide the basis for allowing fast matching of the conditions. For these reasons, we used the modified De Jong & Spears representation (De Jong & Spears, 1991) proposed elsewhere (Llorà, Sastry, & Goldberg, 2005) which presents no byproducts as the problem size increases. The encoding mapping of the ternary alphabet is as follows

$bit_1$	$bit_0$	$ternary\ value$
0	0	#
0	1	0
1	0	1
1	1	#

For efficiency purposes, as explained below, the 00 case will never be used, and if generated it will be turned into the 11 case. Such a decision will allow the implementation of fast matching bit-level matching strategies. An interesting byproduct of this encoding is that a *xor* operation of  $bit_1$  and  $bit_0$  will indicate if a given conditions is specific or general—useful for some XCS behavior models. It also presents a 1:1 specific to general ratio. This rule encoding also requires the instance attribute to be encoded in a similar manner, as shown below

$bit_1$	$bit_0$	$instance\ value$
0	0	n.a.
0	1	0
1	0	1
1	1	n.a.

Only two case are required to encode the binary instances.

### 3.3 Bit level matching

The encoding presented above allows implementing the condition matching using two bit-wise operations. First, compute the logical *and* between the condition rule and the instance attribute, then, compare the result with the original instance attribute value. If both values are *equal*, then the condition has been matched. However, a rule contains multiple conditions that need to be check. Current CPUs provide logical operations for 32-bit unsigned integers, allowing packing up to 16 conditions per integer—do not wasting any memory. Moreover, performing the matching process described at the unsigned integer level will provide the parallel matching of 16 conditions at once. Figure 3 presents a possible implementation of the bit-wise matching using integer blocks.

## 4 Parallel matching with vector processing

The previous section presented how up to 16 conditions can be packed together and treated as unit to perform parallel matching. This raises the question of how far we can take this approach. This section reviews widely available vector instructions sets and how they can help packing even more conditions into parallel matching units.

Figure 3: Default unsigned integer-based rule implementation and matching. The commented code is a simple improvement to the linear exploration of conditions. The full code can be found at <http://www.uiuc.edu/~xllora>.

---

```
#define RECODE_BLOCKS (2*NUM_CONDITIONS/(8*sizeof(unsigned int))+1)
#define RULE unsigned int*

int isRuleMatched ( RULE rule, INSTANCE ins ) {
    register int i,iFlag;

    for ( i=0, iFlag=1 ;
          i<=RECODE_BLOCKS /*&& iFlag*/ ;
          i++)
        if ( (rule[i]&ins[i]) != ins[i] )
            iFlag = 0;

    return iFlag;
}
```

---

#### 4.1 How much can we chew at once?

Recently, multimedia and scientific applications have pushed CPU manufactures to include again support for vector instruction sets in their processors. Vector instructions can perform parallel logical, integer, and floating point operations. The size of the vector operands determines how many conditions can be packed together in single parallel matching unit. Common available vector instruction sets use 128-bit operands, most often manipulated as packed sixteen 8-bit characters, eight 16-bits integers, or four 32-bit integers/floats—being highly dependent on the architecture and the instruction set of the processor. However, regardless of how the vector is packed, it provides 128 bit registers. This means that up to 64 conditions can be pack and processed at once using vector instructions. Future increases in the operand sizes will only allow us to pack even more conditions per matching cycles and obtain better scalability profiles.

#### 4.2 Hardware support for vector processing

There are several 128-bit vector instruction sets depending on the CPU manufacturer. Motorola/IBM provide AltiVec (e.g. PowerPC G4, G5), Intel provides SSE2 (e.g. P4, Xeons), and AMD provide 3DNow!+ (e.g. AMD Opterons). AMD also provides support for the SSE2 instruction set in their Opteron processors; hence, the rest of this section will only focus on implementing fast matching strategies using AltiVec and SSE2 instruction sets.

A detailed description of both instructions sets, and the processors architectures, are beyond the scope of this paper. Useful architecture, instructions sets, and programming manuals for AltiVec can be found at <http://developer.apple.com/hardware/ve/tutorial.html>; detailed description of the SSE2 instruction set (MMX, SSE, SSE2, and SSE3 extensions) can be found at [http://www.amd.com/us-en/assets/content\\_type/white\\_papers\\_and\\_tech\\_docs/24592.pdf](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf), and the intrinsic C programming instructions are defined in the `emmintrin.h` header. We used `gcc` to generate AltiVec or SSE2.

### 4.3 Vector-based matching implementation

The fast rule matching based on vector instructions builds on the bit-based strategy presented before. It is a natural extension of the integer encoding, packing four 32-bit integers together. It can be regarded as four pipelines doing the matching at once. Vector programming has some requirements that need to be fulfilled. For instance, vector registers need to be explicitly load; once loaded, vector instructions can be executed. If any result needs to be kept, an explicitly store to memory operation needs to be issued.

The Altivec implementation of the matching procedure issues four vector instructions—see figure 4. For each of 128-bit block, two instructions `load` (`vec_ld`) the conditions and the instance values into two vector registers. Then, the vector instruction `vec_and` performs the logical *and* between the conditions and the instance values. The last step of the matching is to check if the entire resulting vector is equal to the original instance (`vec_all_eq`), and update the control flag. Figure 4 also presents the equivalent matching code using SSE2 instructions. The main difference from the Altivec version is that instead of having a single instruction to make the final comparison (`vec_all_eq`) two properly used intrinsic are needed `_mm_cmpeq_epi32` and `_mm_movemask_epi8`.

## 5 The time jury

The ultimate jury of the usefulness of the matching strategies presented above is time. This section presents the time measures for the three strategies proposed: character-based, bit-based, and vector-based matching. The time measures are analyzed and speedup computed. The speedup introduced by using vector-based matching reaches values up to 96, allowing reasonable times for very large problem sizes, not approachable via the character-based strategies.

### 5.1 Platform and experimental design

The platform used for performing the time measures was a dual Opteron 242 at 1.6GHz with 2Gb of RAM, the OS install was a Gentoo Linux, and the `gcc` compiler version was 3.3.3 using the `-O3` flag. Time measures were taken for the three available matching strategies. The vector-based matching strategy used the SSE2 implementation presented in figure 4. Given a problem size (number of conditions) fifty experiments were averaged. Each experiment consisted on measuring the time required to match 2,000 instances in a 5,000 rules set. The source code of this experiments can be found at <http://www.uiuc.edu/~xllora>.

### 5.2 The memory element

The Opteron processor has two level caches. The capacity of the L1 cache is 128Kb and the L2 one is 1024Kb. Rule sets having smaller memory footprints will take better advantage of the caching techniques. Figure 5 presents the different footprints of the different encoding strategies. As it will be shown below, packing more conditions into the cache can provide extra benefits for speeding up the matching process.

### 5.3 Time measures

The time measures for the three encoding and matching strategies are presented in figure 6. The measures for the three strategies present initial changing slopes on their linear behavior mainly due the cache sizes. When the problem size becomes large enough, such effects rapidly banish providing stable linear relations between the problem size and the average time spend on the experiment.

Figure 4: Default vector-based rule implementation and matching. The commented code is a simple improvement to the linear exploration of conditions. The figure presents the code for AltiVec architecture (PowerPC G4 and G5), and its translation to SSE2 (Intel P4/Xeon and AMD Opteron). The full code can be found at <http://www.uiuc.edu/~xllora>.

---

```

#ifdef ARCH_SSE2
#include <emmintrin.h>
#endif

#define RECODE_BLOCKS (4*(2*NUM_CONDITIONS/(8*4*sizeof(unsigned int))+1))

#define RULE unsigned int*

int isRuleMatched ( RULE rule, INSTANCE ins ) {
    register int iFlag = 1;

#ifdef ARCH_ALTIVEC
// Matching using AltiVec instruction set
register int i,iMax,tmp;
register vector unsigned int vir,vii;

for ( i=0, iMax=RECODE_BLOCKS/4 ;
      i<iMax /*&& iFlag*/ ;
      i++) {
    tmp = i*4;
    vir = vec_ld(0, &rule[tmp]);
    vii = vec_ld(0, &ins[tmp]);
    vir = vec_and(vir,vii);
    iFlag &= vec_all_eq(vir,vii);
}
#endif

#ifdef ARCH_SSE2
// Matching using SSE2 instruction set
register int i,iMax,tmp;
__m128i vir,vii;

for ( i=0, iMax=RECODE_BLOCKS/4 ;
      i<iMax /*&& iFlag*/ ;
      i++) {
    tmp = i*4;
    vir = _mm_load_si128((__m128i*)&rule[tmp]);
    vii = _mm_load_si128((__m128i*)&ins[tmp]);
    vir = _mm_and_si128(vir,vii);
    vii = _mm_cmpeq_epi32(vir,vii);
    iFlag &= (-1 == _mm_movemask_epi8(vii));
}
#endif

return iFlag;
}

```

---

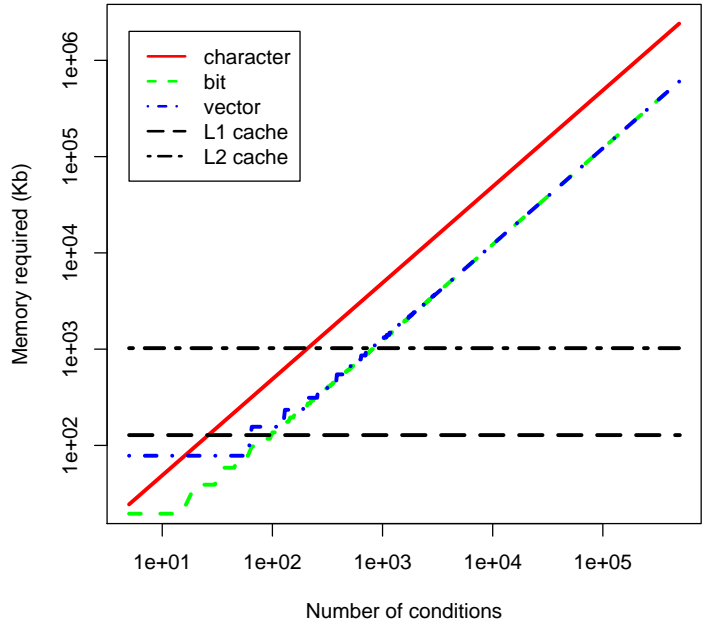


Figure 5: Memory required to allocate a 5,000 rule set as a function of the problem dimensionality.

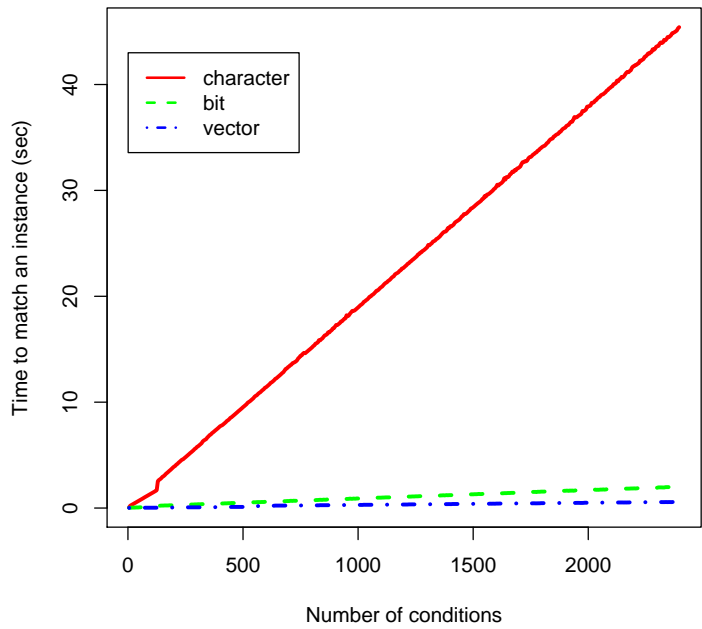


Figure 6: Time required to match 2,000 instances in a 5,000 rule set as a function of the problem dimensionality.

## 5.4 Speedup

The speedup analysis conducted using the time measures—shown in figure 6—is presented in figure 7. The figure presents the speed up of bit-based and vector-based matching compared to the original character-based matching. It also shows the speedup between the bit-based matching and the character-based one. The first interesting observation is the impact of the memory footprints of the different encodings. The bit-based and vector-based compact encodings allow them to take advantage of caching techniques for a longer period. This is reflected in figure 7, where up to 560 conditions the vector-based encoding better exploits cache usage.

Another interesting observation is the speedup for the bit-based and vector-based encoding are above of what theoretically was expected—16 and 64 respectively. This is the result of the efficient implementation of the matching algorithm, using less number of instructions and CPU cycles per matching iteration than the character-based counterpart. This is also supported by the speedup among them, where the speedup tends to the theoretical achievable one—four times due to the vector instructions that operated four integers at once.

To conclude the speedup analysis we wanted to know if the speedup provided by the vector-based implementation was bounded. As show in figure 7 it seemed to grow as we increased the number of conditions. As show in figure 8(a), the speedup could be approximated by  $s(NC) = a_0 \cdot (1 - \exp(-a_1 \cdot NC)) \cdot \log(NC)$ . For that reason we ran the vector-based fast matching using 5,000, 10,000, 100,000, and 500,000 number of conditions 8(b). The new results showed that such an approximation is accurate up to 10,000 rules, but afterwards the speedup saturates to a value of 96 and the approximation is not valid any longer. However, the final speedup is 1.5 larger than what expected initially due to the efficient encoding and fast matching process that reduces the number of instructions and cycles needed.

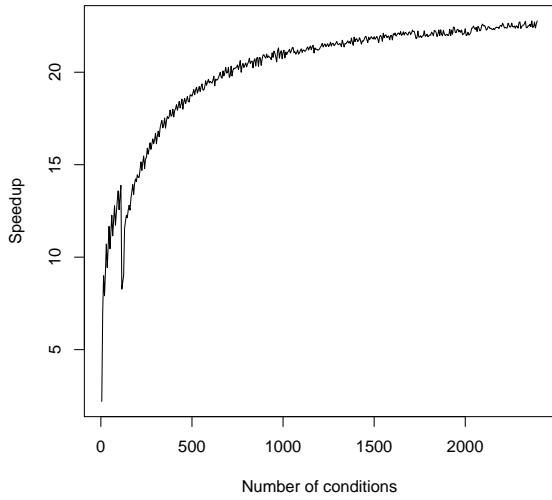
## 6 Conclusions and further work

This paper has profiled two freely available XCS implementations, `xcslib` and `XCS-C`. The profiling results raised the need for (1) reducing the memory footprint of the traditional character-based rule encoding schemes used, and (2) accelerating the faster matching algorithms. We have proposed an encoding that provides a smaller memory footprint and exploits hardware vector operations via AltiVec and SSE2 instruction sets to dramatically reduce the amount of time spend on the matching process. Remarkable speedups were obtained, suggesting great time benefits to any LCS that requires intensive matching stages, as XCS requires, making it possible to deal large number of conditions—500,000 conditions—in a reasonable time.

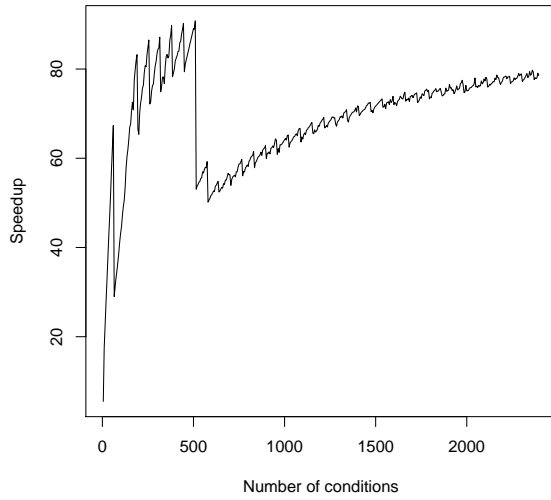
Further work should focus on introducing the proposed matching approaches to some of the freely available XCS implementations and benefit from the reduction of the execution time. Also, the same vector approach can be easily applied to real-valued environments, bringing the benefits of hardware-based parallel processing based on vector operations.

## Acknowledgments

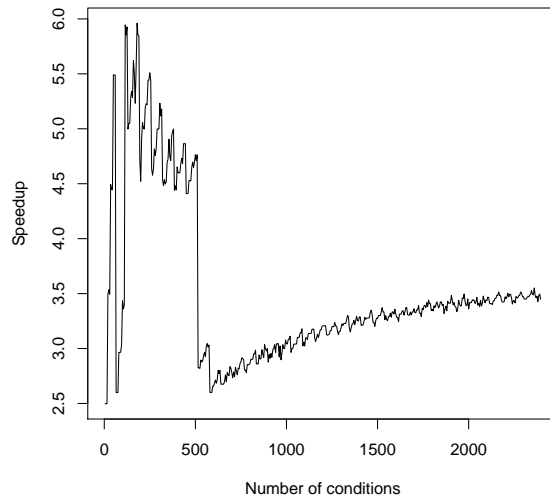
This work was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF (F49620-03-1-0129), and by the Technology Research, Education, and Commercialization Center (TRECC), at University of Illinois at Urbana-Champaign, administered by the National Center for Supercomputing Applications (NCSA) and funded by the Office of Naval Research (N00014-01-1-0175). The US Government is authorized to reproduce and distribute reprints



(a) Speedup of the vector-based encoding versus the bit-based one.

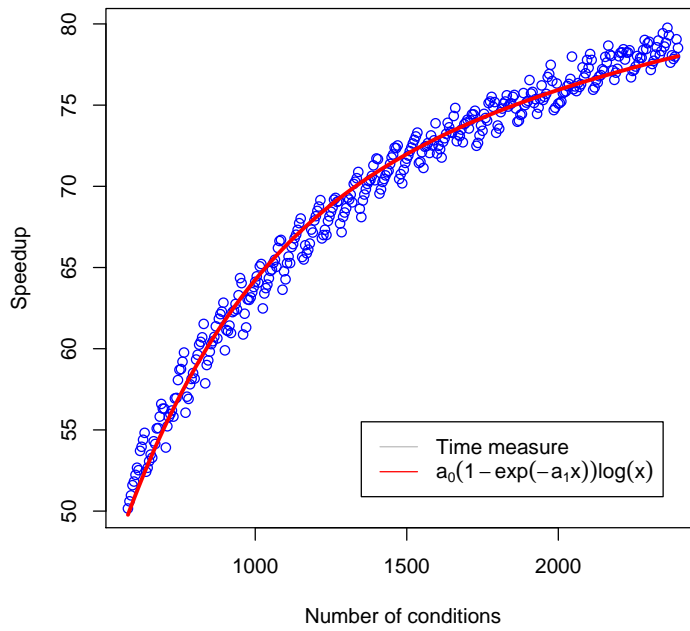


(b) Speedup of the vector-based encoding versus the char-based one.

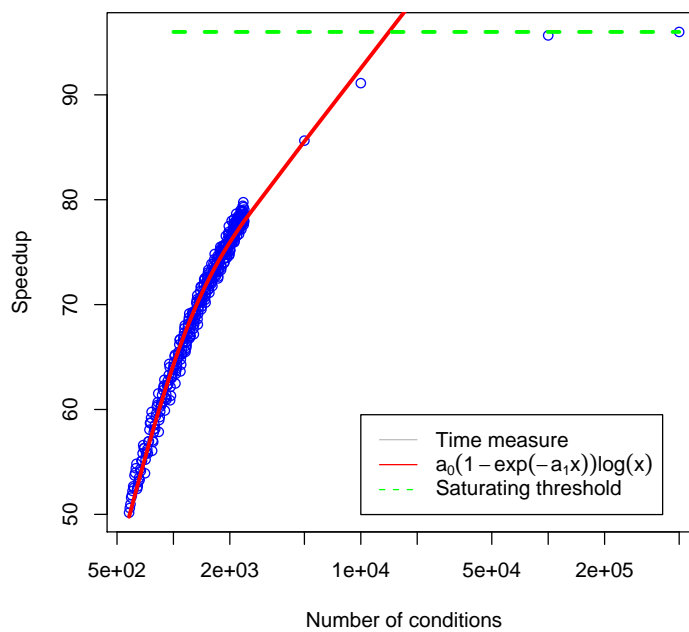


(c) Speedup of the bit-based encoding versus the vector-based one.

Figure 7: Speedup between the three different strategies analyzed. The time is measured used are the ones presented in figure 6.



(a) Speedup growing rate



(b) Speedup saturation threshold

Figure 8: Speedup growing rate analysis. Beyond 10,000 conditions per rule the approximations is not valid any longer and the speedup ends saturating to 96 after 500,000 conditions per rule.

for Government purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the Technology Research, Education, and Commercialization Center, the Office of Naval Research, or the U.S. Government.

## References

- Butz, M., Kovacs, T., Lanzi, P. L., & Wilson, S. W. (2004). Toward a Theory of Generalization and Learning in XCS. *IEEE Transactions on Evolutionary Computation*, 8(1), 28–46.
- Butz, M. V., & Wilson, S. W. (2000). *An algorithmic description of XCS* (IlligAL Report No. 2000017). Urbana, IL: University of Illinois at Urbana-Champaign.
- De Jong, K. A., & Spears, W. M. (1991). Learning Concept Classification Rules using Genetic Algorithms. In *Proceedings of the Twelfth International Conference on Artificial Intelligence IJCAI-91*, Volume 2 pp. 651–656. Morgan Kaufmann.
- Giraldez, R., Aguilar-Ruiz, J. S., & Riquelme, J. C. (2005). Knowledge-based Fast Evaluation for Evolutionary Learning. *IEEE Transactions on Systems, Man and Cybernetics, Part C*, 35(2), 254–261.
- Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*. Reading, MA: Addison-Wesley.
- Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor, MI: University of Michigan Press.
- Holland, J. H., & Reitman, J. S. (1978). Cognitive systems based on adaptive algorithms. In *Pattern Directed Inference Systems* (pp. 313–329). New York: Academic Press.
- Lanzi, P. L. (2005). xcslib: The XCS library. <http://xcslib.sourceforge.net/>.
- Llorà, X., Sastry, K., & Goldberg, D. (2005). The compact classifier system: Motivation, analysis and first results. In *Proceedings of the Congress on Evolutionary Computation*, Volume 1 pp. 596–603.
- Wilson, S. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2), 149–175.