The D-Loop Pattern

A technique for optimizing loops containing conditional branches

Jean-François Maquiné
d-loop@onversity.com

Version 1.10
March 09, 2004
(latest revision : March 27, 2004)

Forewords **(for the 1.1 revision)** :

It seems that some readers of the 1.0x revision are a little bit confused about the main goal of that White-paper. So let me clear some point. The present document is not trying to tell the readers that D-loop is far more better than any other optimizations technics, but try to give a global view of the method. This explains the important mathematical chapter. D-loop is an optimization among others, In some cases it could be better in other cases not. The givens examples are only here to show how to use D-loop, not to suggest there is no other optimization that could be faster.

The revision 1.1 adds a chapter about the practical use of D-loop with several compilers, processors, languages and knows restrictions.

1 Introduction

Modern processors can achieve high execution speeds on ALU instructions, and that's particularity true when using loops. But to reach such speeds, you need to reduce the number of conditional branches and increase the parallelism between instructions.

About conditional branches: One of the most important bottlenecks that may appear in an algorithm is an excessive amount of conditional tests. To understand this, you have to realize how modern processors handle conditional tests. When a modern processor encounters a conditional test, it tries to guess its outcome and execute the rest of the program accordingly; all this before actually knowing the results of the aforementioned test. This mechanism is called branch prediction. It enables the processor to execute the program as if there were actually no test, and virtually eliminate any pipeline stall that can occur when the processor waits for the result of a test.

But that's in theory. In practice, the processor can guess wrong, in which case it has to undo everything executed after the test and that shouldn't have been. To achieve this, the processor removes all the instructions and intermediate results from its pipeline and jumps back to the concerned test before resuming the execution. This process, called "pipeline flush" is extremely expensive in CPU time, especially considering the fact that processors tend to have longer and longer pipeline[1].

About parallelism: Achieving more instructions per cycle (IPC) is what parallelism mean. Apart from increasing the frequency, it's one of the best ways to speed up execution. But it's not an easy task. Neither for processor engineers who try to find new architectures that offer more parallelism capabilities, neither for the programmers who search for algorithms that allow better parallelisms, nor for compilers that ought to schedule instructions in the right order.

In practice, there are lots of reasons why perfect parallelism can't always be achieved and one of them is the occurrence of conditional branches. Conditional branches, even when the prediction mechanism works at its best, will reduce the processors capacity to execute several instructions per cycle.

The technique I describe in this paper allows reducing the average number of tests. It mainly applies to loops doing conditional processing on a sequential structure. This technique, which is called Double Loop or D-Loop, not only reduces the average number of conditional branches per element, but also rise the parallelism of code.

---

[1] The AMD Athlon 64 has 12 stages pipeline, the Intel Pentium 4 Northwood 20 stages and the Intel Pentium 4 Prescott 31 stages

The first part explains the D-Loop technique, analyses it and tries to find out when this method is advantageous and when it's not. Using various examples, the second part explores D-Loop in different situations. All examples are in ANSI C89 but the technique itself can be applied in mostly any language.

Note : You'll find in this white paper two kinds of D-Loop algorithm: a standard and an optimized version. I'll use the former for all examples because it may be simpler to understand, but in practice the latter should be used.

French readers should read Onversity's article about D-loop :

http://www.onversity.com/cgi-bin/progarti/art_aff.cgi?Eudo=bgteob&M=informat&O=touslesmots&P=a0404

2 The first steps leading to the D-Loop pattern

This section explains the rational behind the D-Loop pattern via a concrete example.

2.1 Typical conditional problem

To document the concept of the D-Loop pattern let's begin with a simple procedure that counts the occurrence of a specific char in a string. This method would typically be implemented via a simple loop containing a single conditional assignment, as shown by the following code

```
int count_char(char *val, char ch)
{
        int i = 0;

        while((unsigned char)*val != '\0')
        {
                if((unsigned char)*val == (unsigned char)ch)
                        i++;
                val++;
        }
        return(i);
}
```

Listing **1** : typical implementation of count_char()

2.2 Analyzing and improving

While this implementation is considered as the fastest you can write, it contains a serious bottleneck. In fact, for each character in the string it's doing two conditional tests. During each iteration, a first test, the loop condition, is executed to know whether the end of the string has been reached. Right after this, a second test is performed within the loop to check whether the current character is to be count. Knowing that the null character indicating the end of the string is unlikely to be encountered during most iterations, the first test appears superfluous in most cases. Even on most modern CPU, this operation costs a significant amount of time.

In order to improve this situation, the *count_char()* function was re-implemented, but this time using the D-Loop pattern.

```
int new_count_char(char *val, char ch)
{
        unsigned char       char_mask[256];
         int        i=0;

        memset(char_mask,0,256);

        char_mask[(unsigned char)ch] = 1;
        char_mask[0] = 1;

        while((unsigned char) *val != 0)
        {
                while(char_mask[(unsigned char) *val] == 0)
                {
                        val++;
                }
                if((unsigned char)*val == 0)
                        break;
                else
                        i++;
                val++;
        }
        return(i);
}
```

Listing **2** : implementation of count_char() using the D-Loop pattern

As you can see, the single loop from the previous implementation has been split in two parts : a global loop and an inner loop. The global loop is only responsible for handling the end of the string, removing this burden from the rest of the code. This design can be detailed in four specific steps.

1. The global loop checks whether the end of the string was reached, either at the beginning of the function, or when the inner loop has stopped.

2. The inner loop sequentially runs through the string, discarding most of the characters, and stops whenever a character may be important. In this case, either the character contain in 'ch' and the null characters.

3. Right after the inner loop, there is a security test checking whether the current character indicates the end of the string. If it happens to be true, we bail out from the global loop.

4. Finally, the specific operation, which in the count_char function is to increment a counter, is executed and the program goes back to step one.

The idea is that most characters will be skipped in the inner loop using only one conditional test instead of two.

2.3 Efficiency

Notes : All tests have been done using the latest Intel C compiler (version 8.0) on a Pentium 4 Northwood 3.2 GHz. I used the optimized D-loop version (see chapter 7) with the mask technique (see chapter 4). To evaluate the performance of the algorithm, I have used a long string (79000000 characters), effectively making everything outside of the function negligible.

2.4 Comparison table and analysis

char	occurrences	Standard	D-loop	faster (x)
space	12000000	3.953s	3.703s	1.06
e	7000000	3.172s	2.859s	1.11
d	2700000	2.032s	1.547s	1.31
f	700000	1.828s	1.297s	1.41
W	0	1.693s	1.141s	1.48

In the worst case, when the function counts the number of space characters, the D-Loop version is 1.06 time faster. But what's more interesting is that, in the best case, the difference in speed rises to 1.48 as the number of occurrences decreases. This means that the difference in speed between standard and D-Loop version is not constant.

Also the D-Loop version can be significantly faster for such functions if you know the size of the string. This allows you to use the point break technique (see chapter 5), or if you're already using the mask technique, to reduce the number of conditions. The latter, for example, can happen with a filter function where you have to know whether a string has unauthorized characters.

char	occurrences	**Standard** (with length)	**D-loop** (with length)	faster (x)
space	12000000	4.078s	3.140s	1.29
e	7000000	3.266s	2.438s	1.33
d	2700000	2.078s	1.250s	1.66
f	700000	1.813s	1.000s	1.81
W	0	1.674s	0.843s	1.98

When using the D-Loop pattern instead of a standard algorithm, but having to resort to a complementary technique (e.g. the mask), your function can be up to twice faster. This best case scenario corresponds to the reduction of conditional branches as a consequence of the D-Loop pattern as we will see in the chapter 3.

```
int new_count_char_with_length(char *val, char ch, int len)
{
        int     i = 0,
                j = 0;

        *(val+len-1) = ch;
        while((unsigned char)val[j] != (unsigned char)ch)
                j++;
        while(j++ < len)
        {
                i++;
                while((unsigned char)val[j] != (unsigned char)ch)
                        j++;
        }
        *(val+len) = '\0';

        return(i);
}
```

Listing **3** : implementation of count_char_with_length() using the D-Loop pattern

Note : All the count_char examples are here to show how works D-loop, it doesn't mean that you couldn't find a faster function.

3 Theoretical analysis

In this section we are going to present the theory behind the improvements achieved through the use of the D-loop technique. We are also going to show how this can be generalized and used in other codes.

3.1 First consideration about D-loop

The *new_count_char()* function is faster because the inner loop is able to skip most characters with only one test. Looking at this we can split the domain[2] in two subsets.

> The subset A, whose elements cannot be quickly processed by the inner loop. In our example, this is the null and a specific character.
>
> The subset B, whose elements can be quickly processed[3] by the inner loop, which is also the complementary set of A.

3.2 Having two sets is the Key

The D-Loop technique primarily considers the subset B; using the inner loops it sequentially runs through characters that belong to this subset, leaving the handling of characters from the subset A to the global loop.

Being able to split the domain into two complementary subsets is the core of the D-Loop method. Once these subsets defined, it's only a matter of handling elements belonging to one in the inner loop, and handling elements belonging to the other outside the inner loop but inside the global loop.

Note: The D-Loop pattern inherits its name, Double Loop, from its particular design composed of two loops dividing the domain in two subsets.

---

[2] The count_char() function is using the C type unsigned char. The range of integers covered by this type is typically [0, 255] on most implementations.

[3] In the case of count_char(), no real operation is done in the inner loop. The inner loop simply discards elements belonging to the subset B.

3.3 D-loop *K* value

Depending on the situation, applying the D-Loop technique isn't always possible or simply not such a bright idea. The K value will enable us to decide whether or not the D-Loop pattern should be used.

If S represents the ordered set of data that has to be processed and si the pair (value, position) of the ith element in S, then :

$$S = SA \cup SB \qquad SA \cap SB = \varnothing$$

$$SA = \{ si \mid Value(si) \text{ in } A \} \qquad N = |SA|$$
$$\text{and}$$
$$SB = \{ si \mid Value(si) \text{ in } B \} \qquad M = |SB|$$

$$K = \frac{M}{N}$$

This means that K represents the ratio between the number of elements of S that appears in the subset B and the number of elements of S that appears in the subset A. SB and SA contain all the elements from S, respectively belonging to the subset B and A; M and N are respectively the cardinality of SB and SA.

Thus K gives the ratio of the number of elements that will be handled respectively by the inner loop or by the global loop. The more can be done in the inner loop, the best.

3.3.1 What if K tends to infinity ?

If K -> inf, it means that the inner loop will be able to make big leaps in the given set of data S. Large chunks of S will be quickly processed by the inner loop using only one conditional test. In this case the efficiency of the whole function will tend to be the sole efficiency of the inner loop.

3.3.2 What if K tends to 1 ?

If K -> 1, it means that the number of elements from S belonging to A is the same as the number of elements from S belonging to B. Depending on how these elements are regrouped, the efficiency of the whole function will go from average to worse. In the

worst case, the inner loop is roughly processing one element, then stops, leaving the next element for the global loop which in turn processes another element to go back into the inner loop; which leads to a ping-pong effect between the inner loop and the global loop.

3.3.3 What if K tends to 0 ?

If K -> 0, it means that the inner loop is mostly useless and may seriously impair the efficiency of the whole function. In that case, the two subsets A and B should be reconsidered, and one should wonder whether instead it's possible to simply swap both sets and process the other subset in the inner loop, which would lead to K > 1.

3.4 Further relationship between K and efficiency

The K value gives a way to analyze some properties of the input data. These properties can be analyzed deeper to find out when it may be interesting to use the D-Loop technique.

3.4.1 Efficiency of a typical D-loop function

Based on these properties, we can compute the overall cost of a function using the D-Loop pattern. This cost will be measured in number of conditional branches. In the case of a regular function using one loop containing only one inner condition, the cost per element Ce equals 2 conditional branches: there is the terminal condition of the loop, plus the inner test. If n represents the number of elements to be processed, then the overall cost Co of the function is 2n, which is Ce × n.

In the case of a typical D-Loop pattern, an element will face a different number of tests depending on whether it's processed only within the inner loop. If it's the case, the terminal condition of the inner loop is the only condition that will be executed for this element; otherwise four conditional branches will be done (the inner loop terminal condition, the security test, the value test and the global loop terminal condition). So, in the former case the cost for one element (Wb) equals one conditional branch, and in the latter the cost for one element (Wa) equals four conditional branches. Based on this, we can say that the average cost for one element is :

$$Ce = PK * Wb + (1 - PK) * Wa$$

PK is the probability that an element does not trigger the condition of the inner loop, that is the probability that an element, randomly taken from the input data, belongs

to the subset B. PK and K are related to each other and actually express the same properties but in different ways.

$$n = |SB| + |SA| \qquad K = \frac{|SB|}{|SA|}$$

$$PK = \frac{|SB|}{n} \qquad K = \frac{PK}{(1 - PK)}$$

Let's take, for example, the function *count_char()* and say that a 'space' symbol appears once every 6 characters. Based on this we can conclude that:

K = 5/1 Wb = 1
PK = 5/6 Wa = 4

Ce = (5/6 + 1/6 * 4)
= 9/6
= 1.5

Thus, the average cost for this function following the D-Loop pattern is of 1.5 conditional branches per element. This means that this version executes about 25% less conditional branches than the regular implementation. Further in this document, you'll even see that the D-Loop pattern can be optimized so that W1 equals 3 (in our previous example, this would mean that Ce = 8/6 ~= 1.33).

3.4.2 Deciding whether to use the D-loop pattern

If the cost of a function using the D-Loop technique is greater or equal than the cost of its regular implementation, then it's obvious that the D-Loop pattern should not be followed. Thus, Ce for the D-Loop version should be less than 2.

$$Ce < 2$$
$$\Leftrightarrow (PK * Wb + (1 - PK) * Wa) < 2$$
$$\Leftrightarrow PK * (Wb - Wa) + Wa < 2$$

Using Wb = 1 and Wa = 4

$$\begin{aligned} -3PK + 4 &< 2 \\ -3PK &< -2 \\ PK &> 2/3 \end{aligned}$$

Using the relation that exists between K et PK, we can also express the minimal K value.

$$K = \frac{PK}{(1 - PK)}$$

So if PK has to be greater than two thirds, the K value should be greater than 2. All this means that the size of the subset SB should be at least two thirds the size of the input data. So, if more than 2/3 the input data cannot be solely processed by the inner loop, then the D-Loop pattern should not be used.

Note: Using the optimization explained further in this document, Wa equals 3, which brings the previous equation as PK > 1/2. In this case, it means that at least half the input data should be processed by the inner loop in order to make the D-Loop technique attractive.

3.5 Performance analysis on real hardware

We have used Intel Vtune performance analyzer to gather information about count_char() functions. Vtune uses hardware performance counter to measure how often some specific events happen in the microprocessor. We focused on branches as the main focus of the D-loop technique is to improve this aspect of the performance. We will first look at the worse case results (counting the space character in a string).

Statitics	Standard	D-loop
Time execution	0.40s	0.37s
Mispredicted branch	30.000.000	28.000.000
Cost of mispredicted branches	0,19s	0.17s
Branch prediction rate	90%	84%

Count_char() worse case informations(Space char)

The most important piece of information is the cost of mispredicted conditions. The difference between Standard and D-loop count_char functions is 0.02 seconds, which represents a difference of 66% in execution time. This is caused by two things. First, D-Loop can reduce the number of mispredicted branches, and each mispredicted branch can cost about 10 cycles or more (it depends on the pipeline depth). The second reason is that the cost of a mispredicted branch is, in average, lower with D-Loop than with a standard algorithm. To understand this, you ought to know that the cost in cycles of a mispredicted branch is not a constant one, it depends on what kind of instructions are in the pipeline and on the current state of the pipeline when the misprediction occurs.

D-loop has a lower Branch prediction rate[4], which is normal due to the fact that the decrease in mispredicted branches does not follow the decrease in the number of branches. With the future optimizations in the prediction mechanisms of the next generation of processors, D-Loop should increase its momentum over standard algorithm.

Statitics	Standard	D-loop
Time execution	0.170s	0.120s
Mispredicted branch	0	0
Cost of mispredicted branches	0,0s	0.0s
Branch prediction rate	100%	100%

Count_char() best case informations(W char)

[4]Branch prediction rate = (1 - (mispredicted branches / number of branches))*100

But as you can see the difference in execution time isn't caused by the number of mispredicted branches. So where does it come from? In this case, only the inner loop is executed, that particular piece of code is smaller than the code of the function count_char() and has more parallelism. To document this, here is the assembly code of both functions

Assembly code	
Standard	**D-loop** (inner loop only)

0d : cmp ebx,edx
jnz 14 :
add eax,01h
14 : movzx ebx,BYTE PTR [ecx+01h]
add ecx,01h
test ebx,ebx
jnz 0d :
	4d : movzx eax, BYTE PTR [edi]
	movzx edx, BYTE PTR [esp+eax]
	add edi,01h
	test edx,edx
	je 4d:

Count_char() best case information (W char)

So, to recapitulate, D-Loop can respectively increase the average parallelism in the best case and reduce the number of mispredicted branches in the worse case. In both case the reduction of the amount of conditional branches makes it easier to the processor to manage the code.

4 D-Loop combined to a mask technique

The D-Loop pattern can be applied to a lot of functions. At first sight it's not always obvious, but the main idea is to find a relatively quick test that can split the domain in two subsets. This section explains how the mask technique can be used to achieve this goal.

4.1 The widely used strstr() function

In the C standard library, *strstr()* is a function that searches for the first occurrence of a string in another. One of its common uses is to look up for a given word in a text, and eventually, count all its occurrences. A lot of research has been done around this function, papers and books on the matter spread all over the place. But since *strstr()* has been so much analyzed, is it even possible to implement one that beats the

highly optimized assembly versions from the latest compilers such as ICC[5] and GCC[6]. The following implementation in C will try to do so.

While I will not go into an in-depth analysis of this function - it's not a white paper on *strstr()* -, here is a quick description of the algorithm it's using before we move to the principles of the mask. Str2 is the string we are searching in the string str1.

1. If str2 is empty, we have to return str1, so says the C standard.

2. If str2 contains only one character, use *strchr()* instead.

3. Create a mask, a table, whose each entry respectively corresponds to all possible values of the C type char. All entries are initialized to 0, except the first one (which represents the null character) that is initialized to 1.

4. The inner loop sequentially parses str1 until the entry of the mask for the current character is different from 0.

5. When the inner loop finishes, the current character must be either equal to 0 (the null character) or to the first char of str2. In the former case, no occurrence of str2 could be found in str1 and we return NULL. In the latter case, it's possible that an occurrence was found, this possibility has to be checked.

6. If an occurrence was found then return a pointer indicating its position, else go back to step 4.

4.2 Enabling 2 subset through the use of a mask

The mask is an array of boolean values. It enables us to create the two subsets from the domain we're working on. The first one contains the null character and the first character of the string we're looking for and the second one all the other characters. This technique is already well known and it's used to reduce the number of conditional branches; countless respectable filter functions implements this idea. In the case of the D-Loop pattern applied to the *strstr()* function, this mask is the key to virtually define the two subsets.

---

[5] Intel C compiler (http://www.intel.com)

[6] GNU Compiler collection (http://gcc.gnu.org)

4.3 Efficiency of the D-loop using a mask

By definition, the K value is strongly bound to the input data. Because it's dependent on the content of the two strings, it's either possible to produce an example that misuses the above implementation and that is slower than the standard version, or an example that's two times faster. But these worst and best case scenarios generally don't happen on real-life texts.

According to my own tests, this new algorithm appears to be 1.1 to 1.6 faster than the strstr function give by a compiler. It's an impressive improvement, especially if we take in consideration that an assembly version would be even faster.

To show that the improvement is not due to a specific processor, or compiler or operating system (which doesn't mean it will be fast with any compiler and run fast on any processors, but it's not specific to Pentium 4 Northwood and Intel compiler), I have done two tests. One with Intel compiler 8.0 on a Pentium 4 Northwood running on Windows XP, and one with GCC 3.3.1 on a Pentium 3-S running on Linux.

String (not found)	**Compiler library**	**D-loop**	faster (x)
etat	3.313s	2.890s	1.14
Xayen	1.860s	1.156s	1.60

Windows : Intel C/C++ 8.0. Pentium 4 Northwood 3.2 GHz

String (not found)	**Compiler Library**	**D-loop**	faster (x)
etat	6.984s	5.911s	1.18
Xayen	4.184s	2.857s	1.46

Linux : GCC 3.3.1. Pentium 3-S 1.4 GHz

Note : The strstr() implementation using the D-Loop pattern is based on a simple algorithm. There are more sophisticated algorithms for the strstr() function like the Boyer-Moore [1] algorithm. But such algorithm are not always faster because they need to know the length of the strings and in C it's a costly evaluation.

```
char *new_strstr(const char *str1,const char *str2)
{
	unsigned int		test,
				i,
				k,
				j = 0;

	const unsigned char *st1,
				 *st2;
	unsigned char char_mask[256]		= {
		1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
		0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
		0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
		0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
		0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
		0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
		0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
		0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
	};

	if(*str2 == '\0')
		return((char *)str1);

	if(*(str2+1) == '\0')
		return(strchr(str1,*str2));

	st1 = (const unsigned char *) str1;
	st2 = (const unsigned char *) str2;
	test = (unsigned char) *(str2+1);

	char_mask[*st2] = 1;
	i = 1;

	while(st1[j] != 0)
	{
		while(char_mask[st1[j]] == 0)
		{
			j++;
		}
		if(st1[j] == 0)
			break;
		j++;
		if(st1[j] == test)
		{
			k = j;
			while(st1[k++] == st2[i++])
			{
				if(st2[i] == 0)
				{
					return((char *)(str1+k-i));
				}
			}
			i = 1;
		}
	}
	 return(NULL);
}
```

Listing 4:strstr() implementation using the D-loop pattern

5 Applying D-Loop using a sentinel value

As seen in the previous section, the D-Loop technique sometimes needs to be used in conjunction with another. The function seen in the current section uses a sentinel value technique to apply the D-Loop pattern.

5.1 Big numbers addition

The function introduced here adds the mantissas of two floating point numbers; it was taken from a library created with the sole purpose of computing Pi. In this library, a mantissa is encoded as an array of integers.[7] The first cell gives the size of this array and the regular mantissa binary representation is split into 30 bits blocks stored in the other cells. The remaining two bits in each cell are used for the overflow that will occur during some operations (e.g. during an addition, the 31st bit will be used as a carry flag). The first version[8] (see listing 5) uses a standard approach while the second version (see Listing 6) is based on the D-Loop Pattern.

```
void add(unsigned int * val, const unsigned int * val1 ,const unsigned int * val2)
{
        unsigned int        n = *val1,
                            r = 0,
                            a;

        for (a = 1; a <= n;    ++a)
        {
                val[a] = val1[a] + va2l[a] + r;

                if (val[a] > 0x3fffffff)
                {
                        val[a] &= 0x3fffffff;
                        r =1;
                }
                else
                {
                        r = 0;
                }
        }
}
```

Listing **5**: standard approach to implement add()

[7] The given implementation assumes that an integer has a size of 32 bits. Depending on the platform, the C type *unsigned int* may be larger than this; if this happens the additional bits will be ignored.

[8] Preconditions : (val1[0] == val2[0]) && (val != val1) && (val != val2)

```
void add(unsigned int * val, const unsigned int * val1, const unsigned int * val2)
{
	unsigned int n = *val1,
		b = 0,
		a;

	for (a = 1; a < *val; ++a)
	{
		b += val1[a] + val2[a];
		while (b <= 0x3fffffff)
		{
			val[a] = b;
			a++;
			b = val1[a] + val2[a];
		}
		val[a] = b & 0x3fffffff;
		b = 1;
	}
	val[a] = 0x40000000;
}
```

Listing **6**:Add() implementation using the D-loop pattern

5.2 The sentinel value

As you already know, the D-Loop technique relies on two subsets to work. In this example, these two sets are defined by the couple (b1, b2) where b1 and b2 are the two blocks currently being added. Whether such a couple belongs to the first or to the second subset depends on the result of this addition. In some situations the carry flag has to be set, but not in the others.

But by default, based on the sole result of an addition, the inner loop cannot know when the end of the array is reached and may enter into an infinite loop, which will eventually lead to a segmentation fault. To change this behavior, we use a sentinel value (0x40000000) at the end of each mantissa. This ensures that the inner loop will implicitly detect the end of the input data.

6 Applying D-Loop to single loops

Sometimes, if we can comply with more restricting preconditions, it's possible to use the D-Loop technique even on simple loops. In this paper, a simple loop is defined as a loop not containing any conditional branches except its own terminal condition.

6.1 Another widely used function, strlen()

The function *strlen()* from the C standard library is often implemented using a simple loop (See Listing 6). It returns the length of a string.

```
size_t strlen(const char * src)
{
        const char * src2 = src;

        while (*src2 != '\0')
                ++src2;

        return (src2 - src);
}
```

Listing **7**: standard implementation of strlen()

But since in theory the D-Loop pattern only applies to loops containing conditional branches, how can we use it to such a function? To answer this question, let's implement *strlen()* using the kind of loop expected by the D-Loop pattern (see Listing 7).

6.2 Working around the simple loop

```
size_t new_strlen(const char * src)
{
        const char *src2 = src;

        while (*src2 != '\0')
        {
                if (*src2 == '\0')
                        break;

                ++src2;
        }

        return (src2 - src);
}
```

Listing **8**: strlen() implementation using a loop containing a conditional branch

Now, as it is, the *new_strlen()* function is useless. But it fits the requirements of D-Loop as it defines two subsets: the input characters are either equal to the null character or not. The problem is that if we simply use the D-Loop method the resulting implementation will probably be slower than the first version. So this idea is to speed up the inner loop by reinforcing some preconditions (see Listing 8).

6.3 Stronger requirements

```
size_t new_strlen(const char * src)
{
	unsigned int	car,
			 i = 0;

	car = (unsigned char) src[i] & (unsigned char) src[i + 1];
	i += 2;

	for (;;)
	{
		while (car != 0)
		{
			car = (unsigned char) src[i] & (unsigned char) src[i + 1];
			i += 2;
		}

		if (src[i - 2] == '\0')
			return (i - 2);

		if (src[i - 1] == '\0')
			return (i - 1);

		car = (unsigned char) src[i] & (unsigned char) src[i + 1];
		i += 2;
	}
}
```

Listing **9**: strlen() implementation based on the D-loop pattern

In this new version, we test in the inner loop two consecutive characters at the same time. Using a bitwise logical *and* operation we can quickly know whether one of the two characters may be a null character. After the inner loop, more specific tests enable us to know whether it's the first or the second character, or if it's simply a special case where (a AND b) equals zero without either operands being the null characters.

Now, this D-Loop version of *strlen()* won't always work on all kind of inputs. In fact it brings a new precondition that requires that the string is stored in an array whose size is a multiple of two. Because it tests characters two by two, the function may generate a segmentation fault if this requirement is not met. I've included this function mainly to show you how the D-Loop pattern can be used. But if you need a fast *strlen()* implementation you can always use it (which doesn't mean it's the most faster strlen() function that exist), keeping in mind that you'll need to make sure your string always feature an even size.

Note : We did not transform the strlen() function using the d-loop pattern. But we use (the spirit of) the d-loop pattern to create a new algorithm. So the knowledge that you might acquire with the d-loop pattern can help to create news algorithms that are faster than those existing.

6.4 Efficiency

According to our own tests, timed at 1.2 seconds instead of 2.0, the speed of the D-Loop implementation is 1.66 times the speed of the one shipped with the Intel C Compiler 8.0.

7 Optimizing the D-Loop pattern

The examples given in the previous sections show how one can improve some functions using the D-Loop technique. Although those new implementations are faster, they can still be tweaked and improved a little further.

7.1 Removing the explicit security test

In its current state, one of the weaknesses of the D-Loop pattern is located right after the inner loop: when this loop terminates, the processor encounters three conditional tests. The security test, a value specific test and the terminal condition of the global loop, which is most of the time the same test as the security test.

During the correction of this white paper, Patrice Arrighi pointed to me that the new_count_char() function could be rewritten in such a way that the redundant security test is actually removed. Doing so improves the efficiency of the function as the number of considered tests goes from three to two; in practice a speed up from 0 to 10% is observed, mainly depending on the input data.

The main change in the layout of the D-Loop pattern is that the inner loop now comes after the specific tests. This allows merging the security test with the terminal condition of the global loop. The following listings show the previously seen D-Loop functions rewritten with this optimization in mind.

Note : Because this security condition is always false except when getting at the end of the input data, modern processors using their advanced prediction mechanisms is able to avoid any pipeline flush and thus should virtually skip this test. So this improvement

comes from the fact that the compiler has easier time optimizing this simpler code and also that the processor doesn't waste an ALU on executing this test.

```
int new_count_char(char *val,char ch)
{
        unsigned char       char_mask[256];
        int car,
                i = -1;

        memset(char_mask,0,256);

        char_mask[(unsigned char)ch] = 1;
        char_mask[0] = 1;

        car = (unsigned char) *val;
        while(car != 0)
        {
                i++;
                car = (unsigned char) *val++;
                while(char_mask[car] == 0)
                {
                        car = (unsigned char) *val++;
                }
        }
        return(i);
}
```

Listing **10**: new_count_char() using an optimized d-loop pattern

```
size_t new_strlen(const char * src)
{
        unsigned int i = 0;

        while ((unsigned char) src[i] != '\0')
        {
                if ((unsigned char) src[i + 1] == '\0')
                {
                        I++;
                        break;
                }
                i += 2;
                while (((unsigned char) src[i] & (unsigned char) src[i + 1]) != 0)
                        i += 2;
        }

        return i;
}
```

Listing **11**: new_strlen() using an optimized d-loop pattern

```
char *new_strstr(const char *str1,const char *str2)
{
        unsigned char       char_mask[256];

        unsigned char       *st3;

        unsigned int        car,
                                                    i;

        if((unsigned char)*str2 == '\0')
                return((char *)str1);

        if((unsigned char)*(str2+1) == '\0')
                return(strchr((char *)str1,(char)*str2));

        memset(char_mask,0,256);

        char_mask[(unsigned char)*str2] = 1;
        char_mask[0] = 1;


        car = (unsigned char) *str1++;
        while(char_mask[car] == 0)
        {
                car = (unsigned char) *str1++;
        }
        while(car != 0)
        {
                car = (unsigned char) *str1++;
                if(car == (unsigned char)str2[1])
                {
                        st3 = (unsigned char *) (str1-2);
                        i = 1;
                        while((unsigned char)st3[i] == (unsigned char)str2[i])
                        {
                                i++;
                                if((unsigned char)str2[i] == 0)
                                {
                                return((char *)(str1-2));
                                }
                        }
                }
                while(char_mask[car] == 0)
                {
                        car = (unsigned char) *str1++;
                }
        }
        return(NULL);
}
```

Listing **12**: new_strstr() using the optimized d-loop pattern

8 D-loop and multi-threading

The D-Loop pattern itself can be used on concurrent programming (cf. multi-threading), but complementary techniques might not. For example, if you are using the sentinel value technique, D-Loop can be used in a multi-threading environment and be called by multiple threads at the same time. However, the mask technique cannot always meet all the conditions for this. The actual code you can find in this white paper can be used in such an environment, but be aware that if you decide to use the 'static' keyword in the declaration of the mask it won't be thread-safe anymore (using 'static' may improve the performance in some cases).

The sentinel value technique works as long as you don't evaluate data that overlaps. If you do, it's more than probable that the functions that are called at the same time will shoot into each others legs. But by definition, thread-safe code doesn't imply that you can work on the same piece of memory at the same time.

9 D-Loop : Practical use and restrictions

D-loop is not a optimization that will works in any cases. So I will give here some practical information that can help you to decide to go with D-loop or not. You have made test on specific compiler , processor,... and get bad or good result, let me know.

Compiler

D-loop need that the compiler is able to optimize ALU instructions. That the case for the Intel compiler and GCC, but not for Visual C/C++ version 5 and 6. I do know nothing about version >= 7.x.

The compiler should be able to optimize 'while' loop. That mostly the case for C/C++ compilers, but not for Delphi.

Processor

I've developed and mostly tested D-Loop on an Intel Pentium 4 Northwood (P4C). All variants of the D-Loop technique work very well on this processor, and I suspect on Prescott (P4E) too, due to their common Netburst architecture. In fact, all processors seem to react very well, with only one exception: when using the D-loop with mask, the Athlon-XP badly performs. For example, with the D-Loop new_count_char() function (using a mask) is 9% slower in the worse case than count_char(), while the

new_count_char_with_length() function (not using a mask) is 22% faster in the same case.

About the mask technic. Some readers thinks that it works well only with a fast L1 data cache. Sure a slow latency of the L1 data cache can reduce the performance of the mask technic, but Pentium 3 processor are doing well even with a slower latency than the Pentium 4 Northwood. So the bad performance of the Athlon XP seems not to come from it's latency. If anybodies have an explication, let me know.

I would be happy if someone could make test on Athlon-64 and Prescott processor, specially the optimized Strstr() function

Languages

D-loop seems to like C/C++ language. I don't have any other information about other languages other that Delphi doesn't like while instruction. If you have some let me know.

Inline

D-loop functions are often bigger. This could make inline impossible and/or brings some performance trouble.

If you have any comment to the practical use of D-loop, let me Know, I will be happy to add them.

10 Last words and conclusion

I've been working with the D-Loop technique for quite a while now. And the more I discover about it, the more this optimisation could be part of the feedback directed optimizations used in some compilers that use execution profiles to optimize further the code. Of course I'm not absolutely sure it's feasible in practice, but people in the matter should probably investigate this opening.

Optimizations are important, even if deeply optimizing a program can reveal itself to be a difficult task and extremely expensive in both time and money. In fact, developers tend to rely on the raw power of newer computers to speed up their

applications; which I believe isn't always a good thing. But D-Loop is a simple optimization, relatively easy to implement, and can bring some serious improvements in execution speed. I sincerely hope this white paper will help you to go into that direction.

11 Acknowledgments

I would like to thank Patrice Arrighi for his optimizations to the D-Loop pattern. I would also like to thank Tanguy Fautré for his help on translating this document in English and for the accuracy of his remarks that enhanced the quality of this white paper. I also thank them for pushing me into making a more complete theoretical analysis.

12 References

[1] **Robert Sedgewick** :
Algorithmes en langage C (french edition of : Algorithms in C)
Pages : 291 - 308 (ISBN:2100053310)

[2] **Dan Richard Kaiser** :
Loop optimization techniques on multi-issue architectures
http://citeseer.nj.nec.com/cache/papers/cs/24653/http:zSzzSzwww.eecs.umich.eduzSz~tnmzSztheseszSzdank.pdf/loop-optimization-techniques-on.pdf

[3] **Cheng Yun, Kai Chiu Kwan, Chan Yu Lai** :
Branch prediction strategies using instruction cache
http://citeseer.nj.nec.com/cache/papers/cs/11051/http:zSzzSzturing.ece.wisc.eduzSz~kwanzSzreport_752.pdf/branch-prediction-strategies-using.pdf

[4] **David L.Lilja** :
Reducing the branch penalty in pipelined processors
http://www.elet.polimi.it/upload/gpalermo/Web/WebElet/Didattica/Papers/BP-Survey.pdf