LOGIC OPTIMIZATION ON A CONCURRENT PROCESSING COMPUTER

F. Theeuwen
Department of Electrical Engineering
room EH 7.28
P.O.Box 513
5600MB Eindhoven, The Netherlands
Email:frans@es.ele.tue.nl

ABSTRACT

Optimization of combinational logic plays an important role in the automatic synthesis of integrated circuits, often called Silicon Compilation. The optimization program rewrites a given set of boolean expressions in such a way that, after mapping the expressions onto a set of library cells, an optimal result will be obtained. This optimality can be in terms of area (number of transistors), speed, power consumption or a tradeoff between them. Particular controllers are suitable to be optimized with such a program. A number of packages are developed and described in literature [Thee], [Bar], [Bray]. One of the problems with those packages is the long computer runtime needed for large real world problems. This paper describes the implementation of a logic optimization package described in [Thee] on a supermini computer (The Alliant FX–8) which is able to execute a job concurrently on a number of CPU’s. For our application is was necessary to rearrange the program flow in such a way that large parts of the computational work can be done concurrently by different CPU’s. To be able to run the program in parallel successfully the next conditions have to be fulfilled:

1. The time consuming program parts have to be executed in parallel. This requires that these parts have to be splitted up in a number of subtasks that can be executed (nearly) independent from each other.
2. The memory management must be able to support concurrent processes.

The paper describes the datastructures and the memory functions that are developed, also details about the implementation of the concurrent parts of the program will be given. A table showing promising results is given.

1. INTRODUCTION

Optimization of combinational logic plays an important role in the automatic synthesis of integrated circuits, often called Silicon Compilation. The optimization program rewrites a given set of boolean expressions in such a way that, after mapping the expressions onto a set of library cells, an optimal result will be obtained. This optimality can be in terms of area (number of transistors), speed, power consumption or a tradeoff between them. Particular controllers are suitable to be optimized with such a program. A number of packages are developed [Thee], [Bar], [Bray]. One of the problems with these packages is the long computer runtime needed for large real world problems. This paper describes the implementation of a logic optimization package described in [Thee] on a supermini computer (The Alliant FX–8) which is able to execute a job concurrently on a number of CPU’s. Although the C-compiler of the FX–8 is able to detect concurrency automatic, for our application is was necessary to rearrange the program flow in such a way that large parts of the computational work can be done concurrently by different CPU’s. To be able to run the program in parallel successfully the next conditions have to be fulfilled:

1. The time consuming program parts have to be executed in parallel. This requires that these parts have to be splitted up in a number of subtasks that can be executed (nearly) independent from each other.
2. The memory management must be able to support concurrent processes.

In the next chapters firstly an overview of the computer architecture and the optimization process will be given. In chapter 4 the datastructures and the memory functions will be discussed. Chapter 5 will give details about the implementation of the concurrent parts of the program. In chapter 6 benchmark results will be given.

2. THE ALLIANT COMPUTER

The Alliant FX/8 computer is a supermini computer containing three types of modules:

1. The Computational Elements (CE’s)
   The CE is the computational building block of the system. Each CE is a pipelined processor with integrated floating point and vector instruction sets. The CE’s can be grouped together to operate on one job. Such a group of CE’s is called a complex.

2. The Interactive Processors (IP’s)
   The IP’s are dedicated to execute user interactive jobs, input output functions and operation system functions. They offload the computational complex. The IP’s are dedicated to execute user interactive jobs, input output functions and operation system functions. They offload the computational complex.

3. The Cache and Memory systems The main part of the Cache Memory system is the Computational Processor Cache (CPC). It serves as an high speed physical memory buffer for the computational complex. It allows for:
   1. The use of the data residing in a cache, by each processor, without going through the main memory.
   2. High cycle times in the cache memory avoiding long waiting times if two or more CE’s want to have access to the same portion code or data.
   3. Fast data passing between processors In figure 1 the Alliant architecture is shown.
3. LOGIC OPTIMIZATION

The input to the logic optimization system is a set of boolean expressions. Basically, this set of expressions can be given by a sum of products. To explain the optimization, a few definitions are needed.

3.1 Definitions

1. A variable is a symbol representing a coordinate in the boolean space. For instance, \( a \), \( b \), \( c \), \( d \), \( e \).

2. Variables and their complements are called literals. For instance, \( \overline{a} \), \( \overline{b} \), \( \overline{c} \), \( \overline{d} \), \( \overline{e} \).

3. A cube is a product of literals, such that it contains a variable or its complement. For instance, \( a \cdot \overline{b} \cdot c \cdot d \cdot e \).

4. A boolean expression is a sum of cubes. For instance, \( a \cdot b \cdot c + d \cdot e \).

5. Weak division \( \langle f \rangle \) of \( f \) by \( g \) is defined as the largest set of cubes common to the result of dividing \( f \) by each cube of \( g \). For instance: \( (a \cdot b \cdot c + d \cdot e) / (a + b \cdot c + d \cdot e) \).

6. If \( \langle f \rangle \), \( g \) holds, \( g \) divides \( f \) evenly. For instance: \( a \) divides \( a \cdot b \cdot c + d \cdot e \).\( a \) evenly.

7. A function \( f \) is cube free, if and only if it divides \( f \) evenly. For instance: \( a \cdot b \cdot c + d \cdot e \).\( a \) evenly.

8. A primary divisor \( \langle f \rangle \) of an expression \( f \) is a cube or a subexpression that results from dividing \( f \) by a cube. For instance: \( b \cdot c + d \cdot e \) is a primary divisor of \( a \cdot b \cdot c + d \cdot e \).

9. A kernel is a cube free primary divisor. For instance: \( b \cdot c + d \cdot e \) is a kernel of \( a \cdot b \cdot c + d \cdot e \).

3.2 The optimization

During the optimization process the boolean expression is transformed while keeping their logic functionality. By searching for common subexpressions in the given set of functions, and implementing new intermediate variables for these subexpressions the number of necessary transistors to implement the logic functions can be reduced. During this process also speed and power consumption can be optimized. To perform the optimization, two classes of subexpressions are searched for. The first class of subexpressions are kernels. By repeatedly substituting kernels large common subexpressions can be found. The second class of subexpressions are cubes.

The optimization consists of three steps:

1. Simplification

2. Kernel substitution

3. Cube substitution

This optimization is described in [Bray] and [Thee]. Simplification processes each expression separately and results in a minimal set of prime cubes for each expression. This processes is very suitable to be done in parallel, because each expression can be simplified separately. Each CE can work independently on the simplification of one expression, we will call this "expression parallel" execution. Kernel and cube substitution consist both of four steps:

1. Determination of kernels/cubes, in each expression. This can be done expression parallel, because kernel/cube computation of an expression has no interference with other expressions.

2. Determination of kernels/cubes appearing more than once. This can partly be done in parallel. Comparison of the kernels is no problem because the different kernels are only read. Writing the common kernels to a database is a bit more complicated, and a number of precautions have to be taken here. (see chapter 5.2)

3. Determination of the most favorable kernels/cubes to be substituted. Difficult to implement concurrently.


4. DATA STRUCTURES

Looking at a number of typical applications for the logic optimization package, a number of requirements for the data structure to be used are evident:

1. The number of different literals can be very large (> 1000).
2. A large number of intermediate literals and expressions can result from a rather simple input description.
3. Although the number of literals can be very large the number of literals in one cube is normally rather small.

Because of the above mentioned observations, it is inevitable to design a data structure that adapts in size automatic during the optimization process.

4.1 Set functions

Within the program the expressions are stored as a sum of cubes. To be able to handle cubes with a wide range of literals a flexible set implementation to store the cubes is chosen. A set is built up with a number of subsets. Each subset is able to contain \( \text{NUMBER_OF_SUBSET.getElements} \) elements stored in an array of...
SUBSETELEM = NUMBER_OF_SUBSET_ELEMENTS / sizeof(integer) integers. Each bit in the array represents an element in the subset. Each subset is accompanied with an integer offset. If a subset contains an element \(i\), the element \(i\) is an element of the set.

A set is stored in a linked list of dynamically allocated structures. Each structure contains three fields. The C type definition is given below:

typedef int SUBSET [SUBSETELEM];
typedef struct cubestruct *cube;
typedef struct cubestruct
{
    int offset;
    SUBSET subset;
    cube next;
};

As may be clear from the described datastructure large as well as small sets can be stored economically. Figure 3 shows an example. The set show is \([3,100]\). \(\text{NUMBER-OF-SUBSET-ELEMENTS} = 32\). (an integer is represented with 32 bits)

```plaintext
```

Figure 2. The set datastructure

On this set datastructure a number of set operations are implemented. As an example the code for the function \(\text{setand}\), computing the intersection of two sets, is shown in figure 3.

```plaintext
void setand(resultadress, set1, set2)
    
******
Returns a cube containing literals which are both in cube1 and cube2.
******

***** */ cube set1, set2;
cube *resultadress;
{
    cube helpset, previous;
    int i;
    bool empty;
    if (!*resultadress) NEWSET(*resultadress);
    helpset = *resultadress;
    previous = NULL;
    while (set1 && set2)
    { if (set1->offset > set2->offset) set2 = set2->next;
        else if (set1->offset < set2->offset) set1 = set1->next;
        else
        { if (!helpset) {
            NEWSET(helpset);
            previous->next = helpset;
        }
            empty = TRUE;
            for (i=0; i<NUMBER-OF-SUBSET-ELEMENTS; i++)
                if (helpset->subset[i] = set1->subset[i] &
                    set2->subset[i])
                    empty=FALSE;
            if (!empty)
            { helpset->offset = set1->offset;
                previous = helpset;
                helpset = helpset->next;
            }
            set1 = set1->next;
            set2 = set2->next;
        }
    } dumpset(&helpset);
    if (previous) previous->next = NULL;
    else *resultadress = NULL;
    }
    /* setand */
```

Figure 3. function \(\text{setand}\)

Remark:
If the size of \(\text{NUMBER-OF-SUBSET-ELEMENTS}\) is large enough the for loops use of vector processing can be made.

4.2 memory management

Not only the cubes, but also the expressions, kernels and all the other data are stored in dynamically created structures. Figure 4 shows the structure definition of the structure used to store an expression.

```plaintext
typedef struct_expr_record *expr_psr;
typedef struct_expr_record
{
    ident_type name; /* name of the expression */
    num_expr exprnumber; /* number of the expression */
    int scheduletime; /* max delay before this expr */
    int delay; /* delay of this circuit */
    int reg_delay; /* delay of the register output */
    float reg_value; /* used for shiftback */
    exprset childset; /* set of dependent gates */
    int amountofcubes; /* amount of cubes */
    cube_ptr contents; /* pointer to cubes */
    literal_set union_r; /* all literals in the cubes */
    literal newvariable; /* intermediate expression name */
    literal latchvar; /* register expression */
    literal invertor; /* literal in invertor */
    bool dual_out; /* complemented output */
    kernel_pnt nextkernel; /* pointer to its kernel(s) */
    expr_pnt nextexpr; /* pointer to the next expression */
    expr_pnt inv; /* pointer to inverse expression */
    expr_record;
```

Figure 4. The expression structure

To be able to run a number of parts of the package concurrently functions to allocate and free memory concurrently must be available. The unix functions \(\text{malloc()}\) and \(\text{free()}\) however are not suitable to be called by a number of CE's running in parallel. Another reason for writing new memory management functions is the fact that the program makes heavily use of temporary dynamically allocated (and freed) memory. Because the units calls \(\text{malloc()}\) and \(\text{free()}\) are rather slow, this would result in low performance of the program.
The basic idea behind the implementation is the use of as many arrays of scratch memory as there are CE’s working on the job.

The C type declaration of the Memory array is:

cube Memory[NUMBER_OF_CES][MEM_SIZE]

If a CE needs some dynamic memory the following actions are taken:
1. Start seeking to a scratch array not locked by an other CE.
2. If a free array has been found this array will be locked.
3. If there are free structures in the array present, a structure is taken from the array and the lock is released, otherwise firstly the array is filled with new memory (with callalloc()) and then a structure is taken from the array. While the array is filled, (with the callalloc() function), the locking variable stacklock will be locked.

Figure 5 shows the function newset()

cube newset()
{
    int i, mem_part = 0;
    cube set;
    while (Memlock[mem_part] == LOCKED) mem_part++;
    lock(&Memlock[mem_part]);
    Mem_ptr[mem_part] = i;
    if (Mem_ptr[mem_part]) = set
    Memory[mem_part][Mem_ptr[mem_part]] =
    else
    lock(&stacklock);
    for (i=0; i<HALF_MEM_SIZE; i++)
    Memory[mem_part][i] = (cube)
    callloc(i, sizeof(struct cube struct));
    unlock(&stacklock);
    set = Memory[mem_part][Mem_ptr[mem_part] + 1];
    unlock(&Memlock[mem_part]);
}

Figure 5. The function newset

The above described procedure ensures that CE’s only have to wait for each other if at the same time two or more memory arrays are exhausted (stacklock). If the arrays are large enough (1,000 elements) this occurs rarely.

5. IMPLEMENTATION OF CONCURRENCY

The Alliant operating system provides the function concurrent_call() to implement concurrency (Allii):

concurrent_call(flags, func, n, argc, <argument list>)

where:
flags : specify some special functions of concurrent_call() (not of interest here)
func : the name of the function to be called concurrently
n : the number of time the function func() has to be called
<argument_list> : list of arguments for the function func.

Concurrent_call() calls the function func() n times. Each invocation of the function func() receives as its first dummy argument the number of the iteration it is processing. The remaining arguments must match the remaining arguments of the concurrent_call() function.

The important part of the function responsible for the computation of the kernels is shown in figure 6. As we can see the computation of the kernels is done expression parallel.

/* calculate all kernels */
#endif ALLIANT

concurrent_call(CNCALL_COUNT/CNCALL_NO-QUIT, detkernels, Exprnum);
#endif else
for (i=0; i<Exprnum; i++) detkernels(i);
#endif

/* find common kernels and put then in comkernel_list */
#endif ALLIANT

concurrent_call(CNCALL_COUNT/CNCALL_NO-QUIT,
intersect_kernels, Exprnum, &comkernel_list);
#endif else
for (i=0; i<Exprnum; i++) intersect_kernels(i, &comkernel_list);
#endif Figure 6. Kernel computation

5.2 intersection of kernels

The intersection of kernels is an example where the locking procedure works well. During kernel intersection kernels are compared. If their intersection is again a kernel, this common kernel intersection has to be written to the common kernel list. Compared with the time needed to intersect all the kernels, the time needed to put the common kernel in a list is only small. Therefore the performance of the system will not degrade when the locking procedures are used here. The function intersect_kernels() is called with an offset. This offset indicates the expression in the list of all the expressions List_of_expr whose kernels are compared with the kernels of all the expressions with a larger offset value. Figure 7 shows the function intersect_kernels.

void intersect_kernels(offset, comklist) int offset; comkern_pr *comklist;
{
    kernel_pr kl, k2;
    expr_pr xl, x2;
    comkern_pr comk;

    for (xl=List_of_expr; offset--; xl=xl->nextexpr);
    newcomkern(&inters);
    newcomkern(&inters->contents);
    for (kl=xl->nextkernel; kl; kl=kl->nextkernel)
        if (kl=(Distill_reg_kernels) k1; k1->nextkernel;
            for (x2=xl; x2;
                k2=(x2=x2->nextexpr); x2=x2=x2->nextkernel; NULL
            for (; k2; k2=k2->nextkernel

432
if(setinters(k1->union_r, k2->union_r))
{
    compkernexpr(k1->contents, k2->contents, inters);
    if((inters->amountofcubes >= Min_kernel_size))
    {
        lock(&Lockaddkern);
        add_common_kernels(k1, k2, inters, comklist);
        unlock(&Lockaddkern);
    }
}

dumpcomkern(&inters);
} /* intersect_kernels */

Figure 7. kernel intersection

6. RESULTS

Table 1 shows some benchmark results. A number of examples are both run on a single CE and on a complex of 4 CE's.

<table>
<thead>
<tr>
<th>name</th>
<th>#of literals</th>
<th>#of literals</th>
<th>cpu time (1 CE)</th>
<th>cpu time (4 CE's)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5xp1</td>
<td>309</td>
<td>182</td>
<td>7.8</td>
<td>2.9</td>
</tr>
<tr>
<td>9sym</td>
<td>889</td>
<td>316</td>
<td>68.3</td>
<td>44.2</td>
</tr>
<tr>
<td>ab2</td>
<td>538</td>
<td>189</td>
<td>19.0</td>
<td>8.6</td>
</tr>
<tr>
<td>alu3</td>
<td>298</td>
<td>138</td>
<td>8.5</td>
<td>4.0</td>
</tr>
<tr>
<td>apla</td>
<td>788</td>
<td>296</td>
<td>36.8</td>
<td>10.4</td>
</tr>
<tr>
<td>bw</td>
<td>486</td>
<td>311</td>
<td>25.4</td>
<td>13.9</td>
</tr>
<tr>
<td>co14</td>
<td>197</td>
<td>102</td>
<td>147.4</td>
<td>92.8</td>
</tr>
<tr>
<td>com1</td>
<td>28</td>
<td>28</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>dc1</td>
<td>77</td>
<td>64</td>
<td>1.6</td>
<td>0.7</td>
</tr>
<tr>
<td>dc2</td>
<td>268</td>
<td>167</td>
<td>9.9</td>
<td>3.3</td>
</tr>
<tr>
<td>dk17</td>
<td>400</td>
<td>182</td>
<td>14.4</td>
<td>4.4</td>
</tr>
<tr>
<td>dk27</td>
<td>189</td>
<td>112</td>
<td>4.1</td>
<td>1.5</td>
</tr>
<tr>
<td>duke2</td>
<td>1807</td>
<td>621</td>
<td>171.1</td>
<td>56.4</td>
</tr>
<tr>
<td>f2</td>
<td>40</td>
<td>34</td>
<td>0.4</td>
<td>0.3</td>
</tr>
<tr>
<td>in6</td>
<td>676</td>
<td>415</td>
<td>50.3</td>
<td>14.6</td>
</tr>
<tr>
<td>in7</td>
<td>562</td>
<td>192</td>
<td>12.8</td>
<td>5.7</td>
</tr>
<tr>
<td>m1x1</td>
<td>102</td>
<td>78</td>
<td>1.7</td>
<td>0.9</td>
</tr>
<tr>
<td>m1x2</td>
<td>989</td>
<td>277</td>
<td>80.3</td>
<td>28.7</td>
</tr>
<tr>
<td>m1x3</td>
<td>11247</td>
<td>5549</td>
<td>4074.0</td>
<td>1223.0</td>
</tr>
<tr>
<td>ming</td>
<td>203</td>
<td>148</td>
<td>6.0</td>
<td>3.5</td>
</tr>
<tr>
<td>mish</td>
<td>122</td>
<td>103</td>
<td>3.0</td>
<td>1.2</td>
</tr>
<tr>
<td>radd</td>
<td>345</td>
<td>92</td>
<td>9.4</td>
<td>4.4</td>
</tr>
<tr>
<td>rd53</td>
<td>159</td>
<td>58</td>
<td>5.0</td>
<td>2.7</td>
</tr>
<tr>
<td>rd73</td>
<td>879</td>
<td>234</td>
<td>60.2</td>
<td>28.1</td>
</tr>
<tr>
<td>rd84</td>
<td>2045</td>
<td>358</td>
<td>177.1</td>
<td>84.8</td>
</tr>
<tr>
<td>risc</td>
<td>233</td>
<td>173</td>
<td>9.7</td>
<td>4.0</td>
</tr>
<tr>
<td>sac2</td>
<td>538</td>
<td>256</td>
<td>34.0</td>
<td>11.4</td>
</tr>
<tr>
<td>sqn</td>
<td>255</td>
<td>169</td>
<td>10.2</td>
<td>3.7</td>
</tr>
<tr>
<td>vg2</td>
<td>812</td>
<td>126</td>
<td>17.1</td>
<td>7.4</td>
</tr>
<tr>
<td>wim</td>
<td>74</td>
<td>60</td>
<td>1.0</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 1. Benchmark results

As can be seen the speedup for 4 CE's is sometimes larger than a factor four (prim12). This is caused by the fact that because of the concurrency of the job the order in which the common kernels and the common cubes are found can differ from the scalar version. This can cause other choices of common subexpressions, thus influencing the remaining part of the computation process.

7. CONCLUSIONS

As the results show, the performance of the optimization program is very promising. The code stays portable to other machines except for a few small parts. An average reduction of the time to solution of a factor of 3.5 is obtained. The speedup for large examples is larger then for small examples. This is because in large jobs relatively more time is spend in the concurrent parts of the program. In the future the vectorisation possibilities of the program will be investigated, and some other parts of the program will also be parallelized.

8. REFERENCES