Department of Electrical & Computer Engineering, University of Thessaly, Volos, Greece.

Implementation of Optimizing Transformations in a High Level Synthesis Compiler

Υλοποίηση Βελτιστοποιητικών Μετασχηματισμών Κώδικα σε Μεταγλωττιστή Σύνθεσης Υψηλού Επιπέδου

Thesis by

Georgios Chatzianastasiou

Supervisor: Georgios Stamoulis
Co-Supervisor: Nestoras Evmorfopoulos

Ευχαριστίες

Με την περάτωση της παρούσας εργασίας, θα ήθελα να ευχαριστήσω θερμά τον κ. Γεώργιο Δημητρίου για τον πολύτιμο χρόνο που διέθεσε, για την εμπιστοσύνη που έδειξε στο πρόσωπο μου, την άριστη συνεργασία, την συνεχή καθοδήγηση και τις ουσιώδεις υποδείξεις και παρεμβάσεις, που διευκόλυναν την εκπόνηση της πτυχιακής εργασίας, καθώς και τους επιβλέποντες της διπλωματικής εργασίας κ. Σταμούλη Γεώργιο και κ. Νέστωρα Ευμορφόπουλο, για τις πολύτιμες και εύστοχες συμβουλές για την σταδιοδρομία μου.
Τέλος, οφείλω ένα μεγάλο ευχαριστώ στην οικογένειά μου και στους φίλους μου για την αμέριστη υποστήριξη και την ανεκτίμητη βοήθεια που μου παρείχαν τόσο κατα την διάρκεια των σπουδών μου, οσο και κατα την εκπόνηση της διπλωματικής εργασίας.

Χατζηαναστασίου Γεώργιος
Βόλος, 2016

Contents

Abstract — **High-level synthesis is the technique that translates high-level programming language programs into equivalent hardware descriptions. The use of conventional programming languages as input to high-level synthesis is challenging, due to the conceptual differences between software programs and hardware descriptions, but is nonetheless becoming the preferred input to high-level synthesis tools. Compilers play an important role in this process, since they can bridge such differences, thus making high-level synthesis tools better accepted by the scientific community, but they can also apply code transformations that target an optimized hardware output. In this work, we firstly discuss a number of transformations that were implemented in the C language front end of the CCC high-level synthesis tool and present experiments of such transformations executed by hand, using as benchmark the Mpeg2 open-source C language code. These experiments prove that compiler optimizations can have a significant positive impact in high-level synthesis tools. Furthermore, the main purpose of this thesis work is to automatically apply loop unrolling technique in order to optimize the C language CCC high-level synthesis tool front end and provide better hardware output. Finally, we present the results of this automatic process and explain our modifications in the front end of the tool.**

Keywords—high-level synthesis; formal hardware synthesis; compilers; compiler optimizations; programming languages; hardware description languages; RTL design; loop unrolling; code motion;

1. Introduction

High-level synthesis (HLS) is a hardware design technique that allows architecture specification as a high-level language program. Instead of using hardware description language (HDL) code, high-level programming languages are used to realize abstract algorithmic executable specifications. Most HLS tools apply severe restrictions on the input language, limiting the tools' capabilities and making them less attractive to designers. However, compiler techniques can be employed, in order to relax such restrictions, give programming flexibility to HLS tools, and also optimize output hardware.

In the this work, we are dealing with a C language compiler front-end, which accepts C language programs as input, and feeds the CCC high-level synthesis tool with an equivalent ADA code. The CCC tool translates programs written in the ADA language into VHDL and Verilog [4]. Except for direct C to ADA translation, the tool uses several compilation techniques that subsequently direct CCC to produce better HDL output.

This thesis work is being categorized in six sections, except for the introduction (Section 1), we describe the problem of high-level synthesis and compiler techniques for high-level synthesis, reviewing similar work, in Section 2. We then give a closer look to the C-To-ADA translator, giving some examples that illustrate the way the translation takes place, in Section 3. In Section 4. we discuss compiler transformations that we perform in order to obtain a better final code. Also, in this section we present experimentation with the tool and such transformations, commenting on the results. Then, in Section 5 we explain the automatic loop unrolling process and the modifications that implemented in the front end of the tool, while in Section 6 we give our conclusions, summarizing the whole work and specifying our future work.

2. Background and Existing Work

The most understood and explored HLS tasks are high-level optimizations, scheduling, allocation and binding [1-6]. High-Level Synthesis research commenced in the 80s, with the first academic and industrial, linear processing HLS tools appearing in the early 90s.

HLS tools are still not widely accepted in industry because of their low quality of results, particularly for real applications with complex module/control-flow hierarchy. Most of the available HLS tools impose severe extensions or restrictions on the programming semantic model of the subset that they accept as specification. The most well-known academic or research- based HLS efforts are the SPARK tool [2] which accepts as input a small subset of the ANSI-C language (e.g. while loops are not accepted), and a conditional guard based optimization method [6]. The latter set the basis for optimizing conditional source code at the beginning of the previous decade.

Recent research efforts on HLS include a multi- speculative approach to synthesize complex adders during datapath synthesis, which again contributes only towards linear flow oriented designs [7], a fixed-point accuracy analysis and optimization of polynomial data-flow graphs with respect to a reference model that is found in many DSP applications [8], a technique to improve nested loop pipelining for HLS, called Polyhedral Bubble Insertion [9], an equivalence checking method of FSMs with datapaths based on value propagation over model paths, for validation code motion, usually applied during the HLS scheduling phase [10], a formal method for accurate high-level casting of optimal adders and subtractors [11], and an exploration approach, called Spectral-aware Pareto Iterative Refinement, that uses response surface models (RSMs) and spectral analysis to predict the design quality without costly architectural synthesis procedures [12].

Compiler optimizations have been considered in high-level synthesis for more than a decade. Loop transformations have been studied for the Spark tool, where loop shifting was tested on the Mpeg1 code [13]. Polyhedral loop transformations for high-level synthesis were studied in [14]. A survey of compilation for reconfigurable computing shows that well- understood techniques of loop unrolling and function inlining have renewed interest and are in the focus of research on high-level synthesis for that field [15].

The most popular industrial HLS tools include the Catapult-C from Calypto (previously developed by Mentor Graphics), Cynthesizer from Forte Design Systems. They all accept as input a small subset of the System-C and C++ languages. These tools [17], have very complicated for the average user interfaces, and they are the most expensive of their class. So, these E-CAD systems are inaccessible for most of the small and medium sized ASIC/FPGA design SMEs.

Other commercial or industrial HLS tools [17], are the Symfony C compiler from Synopsys, the Impulse-C from Impulse Accelerated Technologies, the CyberWorkBench from NEC and finally the C-to-silicon from Cadence. These tools are either used only internally by the producing organization, and they are otherwise not well-known amongst the engineering community.

3. C Language Front End

The C-To-ADA translator is based on the front end of a C compiler. Thus, it performs all early stages of a compiler, including lexical, syntactic and semantic analyses. It then produces an intermediate representation in the form of an abstract syntax tree (ast). In order to produce ADA code, the translator traverses the ast and emits ADA statements for each node of the tree.
The translator output language is a rich subset of the ADA programming language. All details of the ADA language requirements and restrictions for the CCC tool are invisible to the C programmer. The translator obeys all such restrictions, when producing the ADA code, but the C programmer is free from those restrictions. Thus, the gap between the actual input specification language and the high-level language that the former is supposed to be, is very narrow in the tool.
Details on the C-To-ADA translator for the CCC tool can be found in [16]. A short description follows in this section.

3.1 Translating C into ADA

The ADA code produced by the C-To-ADA translator is placed within an ADA package. The declaration part of the package contains type and subprogram declarations, whereas the body part of the package contains all subprogram code.
The types and functions that appear in the ADA code correspond to the types and functions of the C source code. The translator creates type names for all non-standard variable types, keeping original names for any named data types, as well as for all functions. Types produced are the ADA universal integer and the ADA boolean for all C integral types, the ADA range for all C enum types, the ADA arrays for all C array types and the ADA records for all C named struct types. Multidimensional arrays are single-dimensional arrays of arrays. Arrays of records are also supported.

In particular, ADA boolean types are optionally produced by the translator after traversing code to determine if a C integral type is used as a boolean rather than an integer.

Subprograms are declared after type declarations. C functions returning void become ADA procedures, the rest become ADA functions. Subprogram parameters are passed either by value or by reference. Since the default passing mode in the C programming language is by-value, the translator optionally uses a specialized algorithm to detect

C-level parameters that are intended to be reference parameters. Depending on the usage of such parameters, they may be declared at the ADA level as in, out or in out parameters. All parameters retain their C names in the ADA code.

For each ADA subprogram, the package body contains the declaration of local variables and the subprogram body code. The translator keeps the original variable names in the output ADA code. The subprogram body code is a sequence of statements. C expression statements are translated into ADA assignment statements. Control statements are translated accordingly.

3.2 Expression Translation

The translator breaks C expressions into subexpressions, to produce simple ADA expressions. Thus, a single C expression is translated into a sequence of ADA assignments. Unless there is a write on a program variable, each assignment writes its output into a temporary variable, to be used in a subexpression that is higher in the ast. Temporaries are named appropriately.

A subexpression involves the application of a single operator on a number of operands. The translation of most such subexpressions is straightforward.

Nevertheless, some exceptional cases are worth to discuss:

- *Boolean operators*: Binary boolean operators ('&&', '||') are always short-circuited in C. Therefore, for each such operator the translator produces code to evaluate the left operand, and an ADA if statement, to evaluate the right operand only if the left operand is not sufficient for the evaluation of the expression. In order to produce simple expressions involving binary boolean operators, the translator optionally suppresses short-circuit evaluation, allowing full evaluation of logical expressions. The unary '!' operator is translated into a not logical subexpression.

- *Pointer operators* ('*', '&'): The only actual support for such operators is for the use of pointers for reference parameters.

- *Array operator* ('[]'): An array operator can have only a single index expression. The left operand of an array operator must be an array object. The ADA code to evaluate the array index expression will come first, with the actual array access following afterwards.

- *Struct field operators* ('.', '->'): In ADA, references to record fields are identical for both records and pointers to records. Therefore, the two struct field operators of C are translated in exactly the same ADA expressions. The left operand of a '.' operator must be a struct object, whereas the left operand of a '->' operator must be a pointer-to-struct object. The right operand of the operator will be a field of the corresponding struct object.

- *Function calls*: All calls are made to named subprograms. Functions are called within expressions, in which case the result is placed into a temporary. Procedures are called within expression statements, in which the call is the top-level expression, i.e. there is no assignment or other operator outside the call. For each call, the translator will first produce ADA code to evaluate the actual arguments, if any, in order from left to right, and then the actual call.

- *Assignments*: All assignments write a value into a variable. The translator will produce ADA code for the evaluation of the right operand, writing it into a temporary. In the case of compound assignments, the translator will also produce code for the operation to be performed before the actual assignment.

- *Aggregates*: Aggregate expressions are only allowed at array and record initialization.

The expression translation is shown in the following example. Given the following C source expression statement:
x = a[0]?f1(99*c[f1(z)]):88;
the corresponding ADA code produced by the C-To-ADA translator will be:

```
TEMPORARY001 := a(0);
if   TEMPORARY001 /= 0 then

    TEMPORARY002 := f1(z);
    TEMPORARY003 := c(TEMPORARY002);
    TEMPORARY004 := 99 * TEMPORARY003;
    TEMPORARY005 := f1(TEMPORARY004);
    TEMPORARY000 := TEMPORARY005;
else
    TEMPORARY000 := 88;
```

```
end if;
x := TEMPORARY000;
```

assuming all types and variables are properly declared.

3.3 Statement Translation

The translation of C statements other than expression statements into ADA statements, obeys the following rules:

- *If statements*: In the translation of an if statement, the condition expression is translated as an independent expression statement, with the top-level form of the expression incorporated in the if statement.

For example, the C code:

```
      if (x > y + z) a = b;
```

is translated into the ADA code:

```
     TEMPORARY001 := y + z;
       if x > TEMPORARY001 then a := b;
     end if;
```

Any possible nested if statements within an else part are not translated into a unified ifelse part, but rather into a truly nested if statement.

- *While loops*: The condition expression is translated into an independent expression statement, with the resulting top-level form of the expression incorporated in the while statement.

- *Do loops*: a C do loop is translated as a while loop, by replicating the loop body code before the loop, in order to make the loop execute its first iteration.

- *For loops*: ADA supports for loops, but in way quite different from the C for loops. Since there is no straightforward translation of a C for loop into an ADA for loop, the translator transforms a C for loop into a while loop. Optionally, the C-To-ADA translator may attempt to translate a C for loop into an ADA for loop, by checking the following properties of a for loop.

i) The first expression must be a simple assignment into an integer variable.

ii) The second expression must be a simple comparison for less, less-equal, greater or greater-equal of the same integer variable against an integer expression, which must not change value during loop execution.
iii) The third expression must be an increment or decrement of the same integer variable, depending on the comparison of the second expression. If the increment or decrement is coded as a compound assignment, then the right operand must be an integer expression that does not change value during loop execution.

Currently, the translator does not check whether the integer expressions involved in the second and the third expressions change their values within the loop body. The programmer must ensure such a property.

- *Switch statements*: A C switch statement is transformed into an ADA case statement. The selection value is integer, and the expression producing that value is translated into an independent expression statement, writing its output into a special integer temporary named TEMPINT. The ADA *case* statement is then produced, beginning with the line:

```
      case TEMPINT is
```

All C *case* codes are translated into ADA *when* codes.

- *Return statements*: If there is a return expression, the translator produces ADA code to evaluate that expression, writing the output value into a temporary. It then produces an ADA return statement, returning that value. Return statements are only allowed at the end of a C function, and each non-void returning function must end with a return statement.

Other C statements are not supported by the translator.

4. Compiler Optimizations & Experiments

Most of the optimizing transformations that have been proposed over the years in the area of HLS are operation level transformations. In contrast, language level optimizations refer to transformations that change the circuit description at the source level, for example, loop unrolling and code motion. Furthermore, other compiler optimizations, such as function inlining, help to reduce latency. Language-level optimizations can be combined, to produce even greater results. In this section we concentrated on loop unrolling, function inlining, as well as code motion.

4.1 Loop Unrolling

Loop unrolling is the process of placing a duplicate of one or more iterations of the loop body at the end of the current loop body. The loop index variable increment (or decrement) is updated as necessary. The goal of loop unrolling is to extract ILP (Instruction Level Parallelism) by reducing the number of loop iterations, thus eliminating the overhead of end of loop tests on each iteration.

Here is a simple example of two-times loop unrolling. Given the initial C code:

```
for ( i = 0; i < N; i++ ) {
  x[i] = y[i] +1;
}
```

we get the following code after unrolling two times:

```
for ( i = 0; i < N; i+= 2) {
  x[i] = y[i] +1;
  x[i+1] = y[i+1] +1;
}
```

We can highlight that, as the index variable increments by two (the unroll factor) in each iteration, the number of iterations executed is being decreased (divided by the unroll factor). For instance, let us assume N = 100, and unroll factor = 2, it is clear that the number of iterations that will be executed is 100/2 = 50 iterations.

4.2 Function Inlining

The overhead associated with calling and returning from a function can be eliminated by expanding the body of the function inline, and additional opportunities for optimization may be exposed as well.

Here is an illustrative example for inlining. Given the initial C code:

```
void main() {
  double f, c;
  for (f=0.0;f<=300.0;f+=20.0) {
    c = ftoc(f);
  }
}

static double ftoc(double f) {
  return (5.0/9.0)*(f-32.0);
}
```

We get the following code after inlining:

```
void main() {
  double f, c;
  for (f=0.0;f<=300.0;f+=20.0) {
    c = (5.0/9.0)*(f-32.0);
  }
}
```

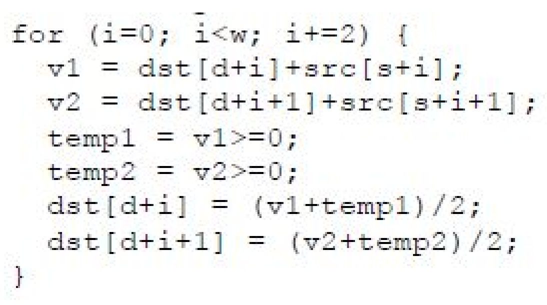
4.3 Code Motion

In some cases, it may be necessary to move instructions inside of a loop body in order to eliminate false dependencies. Code motion except for identifying dependencies, use the technique of renaming variables in order to handle WAW and WAR, avoiding them at the end. In our work, we combine code motion with loop unrolling to increase parallelism.

Let us consider an example from the Mpeg2 open-source code suite that we consider late in this section, specifically in our experiments. The initial code without unrolling and code motion is:

```
for (i=0; i<w; i++) {
  v = dst[d+i] + src[s+i];
  dst[d+i] = (v+(v>=0?1:0))/2;
}
```

The final code after unrolling two-times and code motion is:



```
for (i=0; i<w; i+=2) {
  v1 = dst[d+i]+src[s+i];
  v2 = dst[d+i+1]+src[s+i+1];
  temp1 = v1>=0;
  temp2 = v2>=0;
  dst[d+i] = (v1+temp1)/2;
  dst[d+i+1] = (v2+temp2)/2;
}
```

Except for unrolling and code motion for dependence elimination, we used two temporary variables to replace the comparison expression, in order to further increase ILP. Note how in the above code the two assignments to v1 and v2, the two assignments to temp1 and temp2, as well as the two final assignments to dst[d+i] and dst[d+i+1] can be performed in parallel, something that can be easily detected by the CCC back-end optimizer. Without code motion after unrolling, such parallelism would be impossible to detect in the back-end of the HLS tool.

4.4 Experiments

We can now describe a set of experiments we performed with open-source Mpeg2 C code. In particular, we located code segments that were well-suited for the application of our transformations, and we applied the transformations by hand, evaluating the resulting HDL codes.

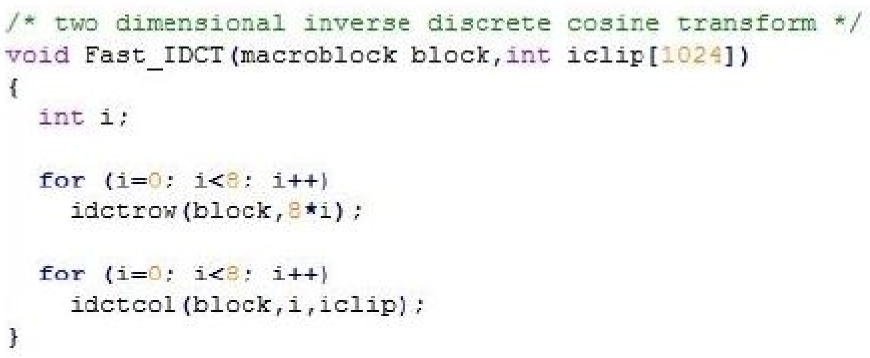
4.4.1 Benchmark Description and Analysis

As a benchmark for testing the HDL code performance, we used the Mpeg2 decoder. We divided the code into small pieces (idct.c , recon.c , spatscal.c , getpic.c) in order to analyze more specifically the features of the code. Briefly, the idct.c was used to implement inlining technique, recon.c and spatscal.c was used for loop unrolling because they include suitable loops. In addition, in recon.c except for loop unrolling we used code motion inside the loops in order to further increase parallelism.

4.4.2 Application of Compiler Optimizations

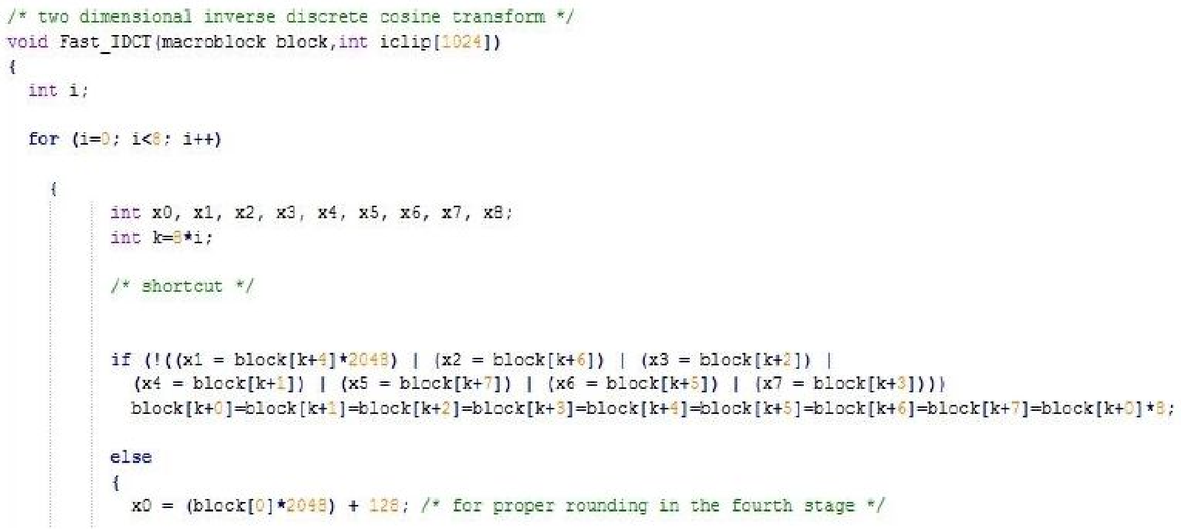
As it was mentioned above, idct.c was the segment of mpeg 2 decoder for inlining implementation. Here we eliminated the overhead of function call and return, by expanding the body of functions: void idctrow (macroblock blk, int ptr) and void idctcol (macroblock blk, int ptr, int iclp[1024]). These functions are being called from void Fast_IDCT (macroblock, int[1024]), where macroblock is an array type of the Mpeg2 code suite.

Snapshots of the Code:: Blocks platform used for handling the C source of function Fast_IDCT() before and after function inlining are shown in Figures 1 and 2, respectively. Similarly, snapshots for the loop spatscal of file spatscal.c that was chosen for unrolling, first the original loop, then the two-times unrolled version are shown in Figures 3 and 4, respectively. Finally, the combination of loop unrolling with code motion, which is the most efficient version of our work, is depicted in loop recon of file recon.c. The corresponding snapshots of the original loop and then the four-times unrolled and scheduled loop are shown in Figures 5 and 6, respectively.



```
/* two dimensional inverse discrete cosine transform */
void Fast_IDCT(macroblock block,int iclip[1024])
{
  int i;

  for (i=0; i<8; i++)
    idctrow(block,8*i);

  for (i=0; i<8; i++)
    idctcol(block,i,iclip);
}
```

Figure 1. Fast_IDCT() before inlining.



```
/* two dimensional inverse discrete cosine transform */
void Fast_IDCT(macroblock block,int iclip[1024])
{
  int i;

  for (i=0; i<8; i++)

    {
        int x0, x1, x2, x3, x4, x5, x6, x7, x8;
        int k=8*i;

        /* shortcut */


        if (!((x1 = block[k+4]*2048) | (x2 = block[k+6]) | (x3 = block[k+2]) |
          (x4 = block[k+1]) | (x5 = block[k+7]) | (x6 = block[k+5]) | (x7 = block[k+3])))
         block[k+0]=block[k+1]=block[k+2]=block[k+3]=block[k+4]=block[k+5]=block[k+6]=block[k+7]=block[k+0]*8;

        else
        {
         x0 = (block[0]*2048) + 128; /* for proper rounding in the fourth stage */
```

Figure 2. Fast_IDCT() after inlining (begin of inlined code).

```
for (i=0; i<lx; i++)
{
  v = 8*(fld0[p0m1+i]+fld0[p0p1+i]) + 2*fld1[p1+i] - fld1[p1m2+i] - fld1[p1p2+i];
  fld0[p0+i] = Clip[(v + ((v>=0) ? 8 : 7))/16];
}
```

Figure 3. Loop spatscal before unrolling.

```
for (i=0; i<lx; i+=2)
{
  v = 8*(fld0[p0m1+i]+fld0[p0p1+i]) + 2*fld1[p1+i] - fld1[p1m2+i] - fld1[p1p2+i];
  fld0[p0+i] = Clip[(v + ((v>=0) ? 8 : 7))/16];

  v = 8*(fld0[p0m1+i+1]+fld0[p0p1+i+1]) + 2*fld1[p1+i+1] - fld1[p1m2+i+1] - fld1[p1p2+i+1];
  fld0[p0+i+1] = Clip[(v + ((v>=0) ? 8 : 7))/16];
}
```

Figure 4. Loop spatscal after 2 × unrolling.

```
for (i=0; i<w; i++)
{
  v = dst[d+i]+src[s+i];
  dst[d+i] = (v+(v>=0?1:0))/2;

}
```

Figure 5. Loop recon before unrolling.

```
for (j=0; j<h; j++)
{
  for (i=0; i<w; i+=4)
  {
    v1 = dst[d+i]+src[s+i];
    v2 = dst[d+i+1]+src[s+i+1];
    v3 = dst[d+i+2]+src[s+i+2];
    v4 = dst[d+i+3]+src[s+i+3];

    temp1=v1>=0;
    temp2=v2>=0;
    temp3=v3>=0;
    temp4=v4>=0;

    dst[d+i] = (v1+temp1)/2;
    dst[d+i+1] = (v2+temp2)/2;
    dst[d+i+2] = (v3+temp3)/2;
    dst[d+i+3] = (v4+temp4)/2;
  }

  s+= lx2;
  d+= lx2;
}
```

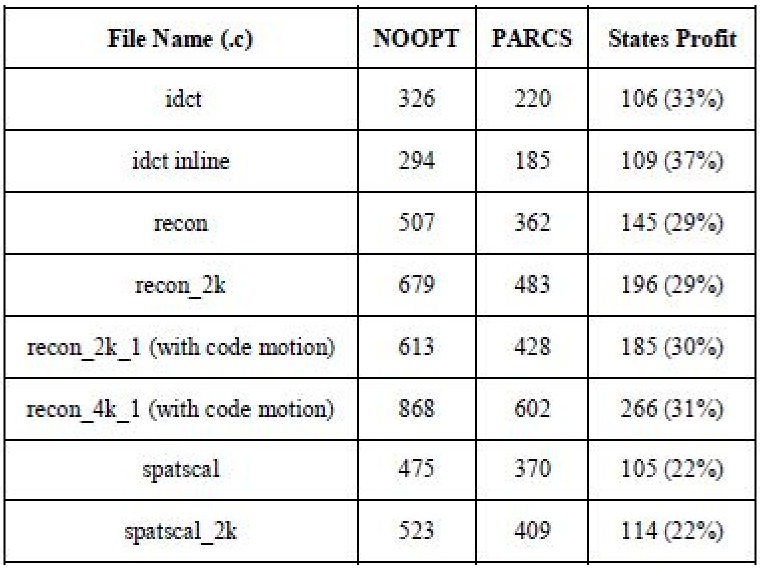
Figure 6. Loop recon after 4 × unrolling and code motion.

4.4.3 HLS: Performance Measurements & Comparisons

The CCC backend provides two versions of HDL code. The first is a non-optimized, whereas the second is an optimized version, produced by the PARCS optimizer of the CCC HLS tool. The PARCS optimizer includes optimizations like code motion, code scheduling, detection and elimination of WAW, WAR dependencies. However, it performs such optimizations at the hardware level, succeeding to exploit ILP at that level, but it cannot deal with source-level dependencies. A compiler-level optimizer can analyze arrays, detect array- access dependencies and optimize code at a high-level, exploiting ILP that PARCS cannot detect.

It is clear that the two tools, the source-level optimizer and PARCS, complement one-another, building a tool that can optimize hardware at the level of state-of-the- art.

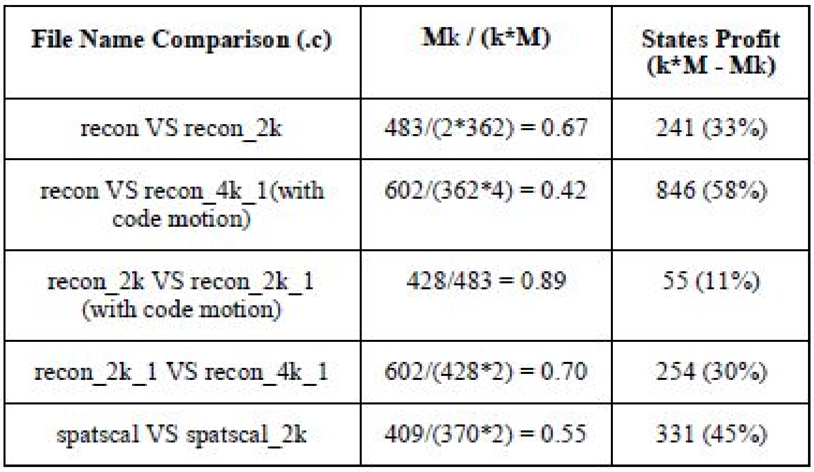
In this section, we first show the states of FSM for each code, comparing the first (NOOPT) version with the PARCS version. Table I gives the results obtained, verifying that PARCS gives a significant reduction in the number of states.



File Name (.c)	NOOPT	PARCS	States Profit
idct	326	220	106 (33%)
idct inline	294	185	109 (37%)
recon	507	362	145 (29%)
recon_2k	679	483	196 (29%)
recon_2k_1 (with code motion)	613	428	185 (30%)
recon_4k_1 (with code motion)	868	602	266 (31%)
spatscal	475	370	105 (22%)
spatscal_2k	523	409	114 (22%)

Table I. Number of States in the VHDL codes.

Next, we compare the coupling of source-level optimized codes with the PARCS optimizer against the plain PARCS optimized codes. For function inlining, we can figure that our profit is the difference in the number of FSM states between idct.c and idct_inline.c, i.e 220-185 = 35 fewer states. For loop unrolling, we made the conservative assumption that in each run, all states are being executed. More specifically, if we suppose N iterations, Mk states after and M states before unrolling, our profit is Mk / (k*M). This number illustrates the reduction of state number in the fully optimized code. Table II gives the profit in state numbers that we obtained. We must note that in actual execution not all states will be executed in each run. However, when using code motion, we will expect that the average Mk against the average M will bring even higher reduction in state number.



File Name Comparison (.c)	Mk / (k*M)	States Profit (k*M - Mk)
recon VS recon_2k	483/(2*362) = 0.67	241 (33%)
recon VS recon_4k_1(with code motion)	602/(362*4) = 0.42	846 (58%)
recon_2k VS recon_2k_1 (with code motion)	428/483 = 0.89	55 (11%)
recon_2k_1 VS recon_4k_1	602/(428*2) = 0.70	254 (30%)
spatscal VS spatscal_2k	409/(370*2) = 0.55	331 (45%)

Table II. State profit when comparing results.

We must clarify that recon_2k is the recon.c file unrolled two times, while spatscal_2k is the spatscal.c file unrolled two times. Also, recon_2k_1 and recon_4k_1 are unrolled two and four times, respectively, and each of them includes code motion optimization. Let us use the second line of Table II to present an example, these figures illustrate a great number of states profit, approximately 60 percent less than the initial number of states for recon.c file. The states before unrolling for recon was M = 362 states, while the number of states after four times unrolling and code motion is Mk = 602 states. The reduction of state number in fully optimized code is Mk /(k * M), where k is the unroll factor, so 602/(362*4) = 0.42. The figure of states profit is being calculated with the mathematical type: (k*M) - Mk, so in our example we have (4*362) - 602 = 846 states profit or 58% improved state number!

Figure 8 gives in graphical form the number of states for the original codes (NOOPT), the PARCS only optimized code, and the combined optimized code (source-level coupled with PARCS).

It is clear, as shown in Table I, that PARCS delivers highly optimized hardware description. It is also clear, as shown further in Table II and Figure 7, that source-level optimizations boost CCC performance, at least in codes where such optimizations are meaningful.

Concerning the optimizations per se, the most significant conclusion is that as the number of k increases, the number of state profit increases too. Thus, loop unrolling confirms its utility and justifies our research. Also, we can figure that, on loop scheduling (recon_2k VS recon_2k_1) in PARCS version, there is no so much difference between the states, because PARCS already includes many optimizations and covers some of the optimizations at source level.

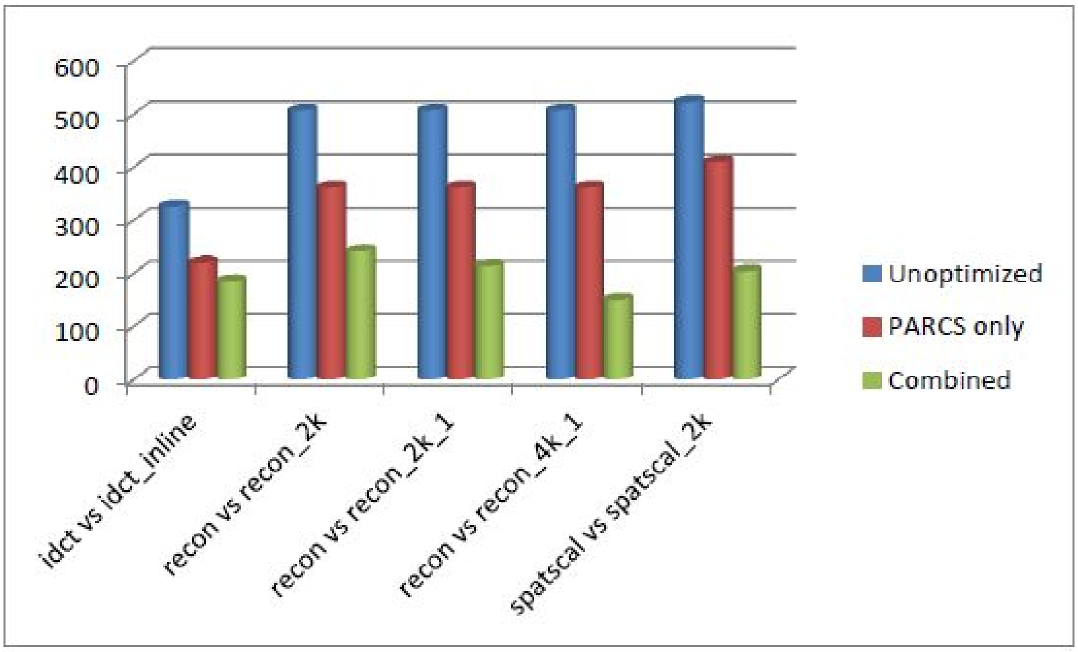


Figure 7. State numbers comparison.

Because the results described in this section were very encouraging, we started to think about finding ways to embed automatic dynamic loop unrolling, enhanced with code motion in the front end of the CCC HLS tool in order to provide better hardware output. This effort is being described in detail in the following section.

5. Automatic CCC Tool's Loop Unrolling Process

In this section we will describe the whole process and the actions which were necessary in order to implement loop unrolling. To achieve this, we have created the appropriate C language code and embedded it into the front end of the CCC tool. It is important to understand that loop unrolling optimization is a source code transformation, as a consequence, the ADA code which is provided by the tool will be updated with the inclusion of the unrolled loops.

First of all, let's give a short description of the tool's intermediate representation (ir). This ir is based on two different types of nodes, the instruction nodes and the expression nodes. The instruction nodes of each subprogram form is a linked list, with arbitrary nested lists for compound statements such as ifs, loops and switches. An instruction node contains among others a 'type' attribute, to define the statement, an 'expression' attribute, to connect to an expression tree associated with the instruction, an 'expr_list' attribute, for statements connected to a list of expression trees rather than a single expression tree, an 'instruction' attribute, to connect to an associated nested instruction node, a 'tail_instruction' attribute, to connect to an alternate nested instruction node, and a 'next' attribute, to connect to the next instruction. An expression node contains among others an 'operator' attribute, to define the expression operation, a 'left' and a 'right' attribute, to connect to two descendants of the node, an 'expr_list' attribute, to connect to a list of expression trees, if the operation has more than two operands, a 'parent' attribute, to connect to the parent node, and an 'instruction' attribute, to connect to the instruction that contains the expression tree.

For instance, the expression:
x = y + 5 ;
is an expression that can be divided into sub-expressions in the way of an abstract syntax tree (ast). The most important of them are the left descendant (the variable x) and the right descendant (y + 5). The left and the right descendants have the same predecessor, the parent node "=". Each expression of an instruction is a part of the ast that can be analyzed further, so as to become leaf of the tree. In our example, the variable x ends up as an identifier leaf, while the right descendant must be analyzed more. So, a graphical representation of our example would be the following:

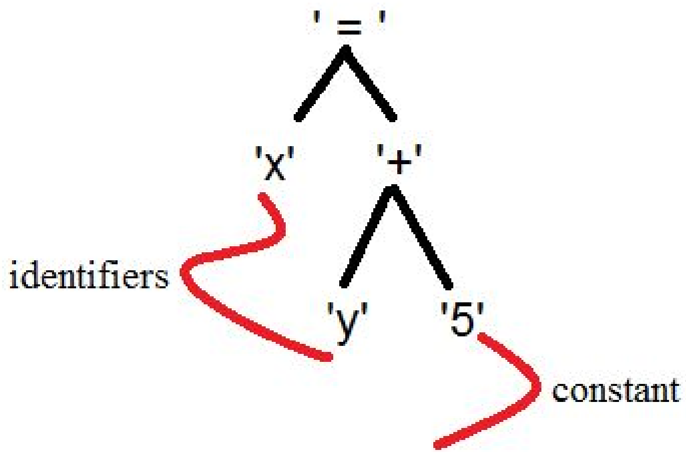
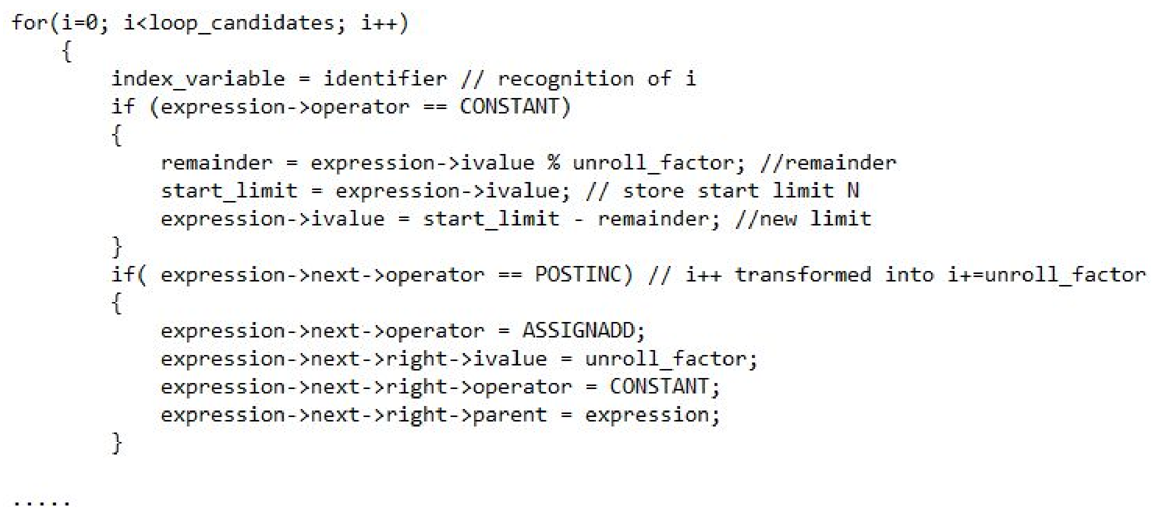


Figure 8. Ast graph of ‘x = y + 5’.

Returning to loop unrolling, we have to clarify that we apply loop unrolling only for loops whose instruction type is FOR_LOOP. So, in order to be able to identify the instruction type, we use the struct “instr_node”. Depending on the instruction type, the attributes of the struct “instr_node” have different values. The most significant attributes for the FOR_LOOP instruction type are ‘expr_list’ and ‘tail_instruction’, the first one illustrates the sequence of the three expressions of the for statement header, while the second one is a pointer that shows the address of the first instruction into the body of the FOR_LOOP.

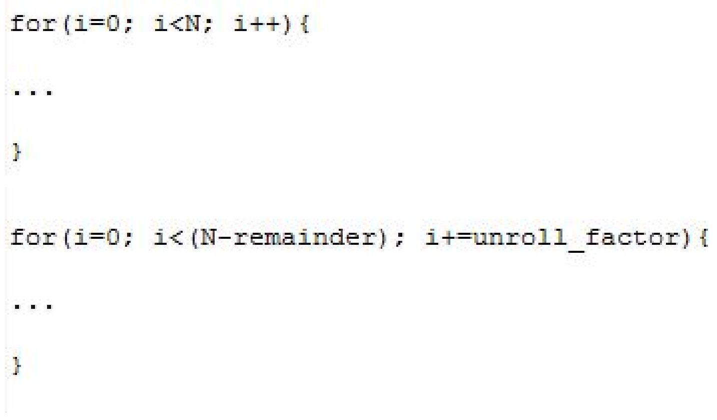
After identifying the suitable FOR_LOOP, for each loop we have to recognize the index variable (the integer which is being incremented on each iteration), and then to store it. In addition, the division of the start limit N of the loop over the unroll factor may give us a remainder. Because of this, we store this number into a variable to help us with the remaining iterations that may still exist after the completion of loop unrolling execution. For the same reason, we change the start limit N into $N - remainder$. Finally, we replace the POSTINC expression (i++) into the ASSIGNADD expression by adding the unroll factor with index variable (i += unroll_factor). The pseudocode of these steps is being presented in Figure 9.



```
for(i=0; i<loop_candidates; i++)
    {
        index_variable = identifier // recognition of i
        if (expression->operator == CONSTANT)
        {
            remainder = expression->ivalue % unroll_factor; //remainder
            start_limit = expression->ivalue; // store start limit N
            expression->ivalue = start_limit - remainder; //new limit
        }
        if( expression->next->operator == POSTINC) // i++ transformed into i+=unroll_factor
        {
            expression->next->operator = ASSIGNADD;
            expression->next->right->ivalue = unroll_factor;
            expression->next->right->operator = CONSTANT;
            expression->next->right->parent = expression;
        }
.....
```

Figure 9. Pseudocode that transforms the header of a FOR_LOOP.

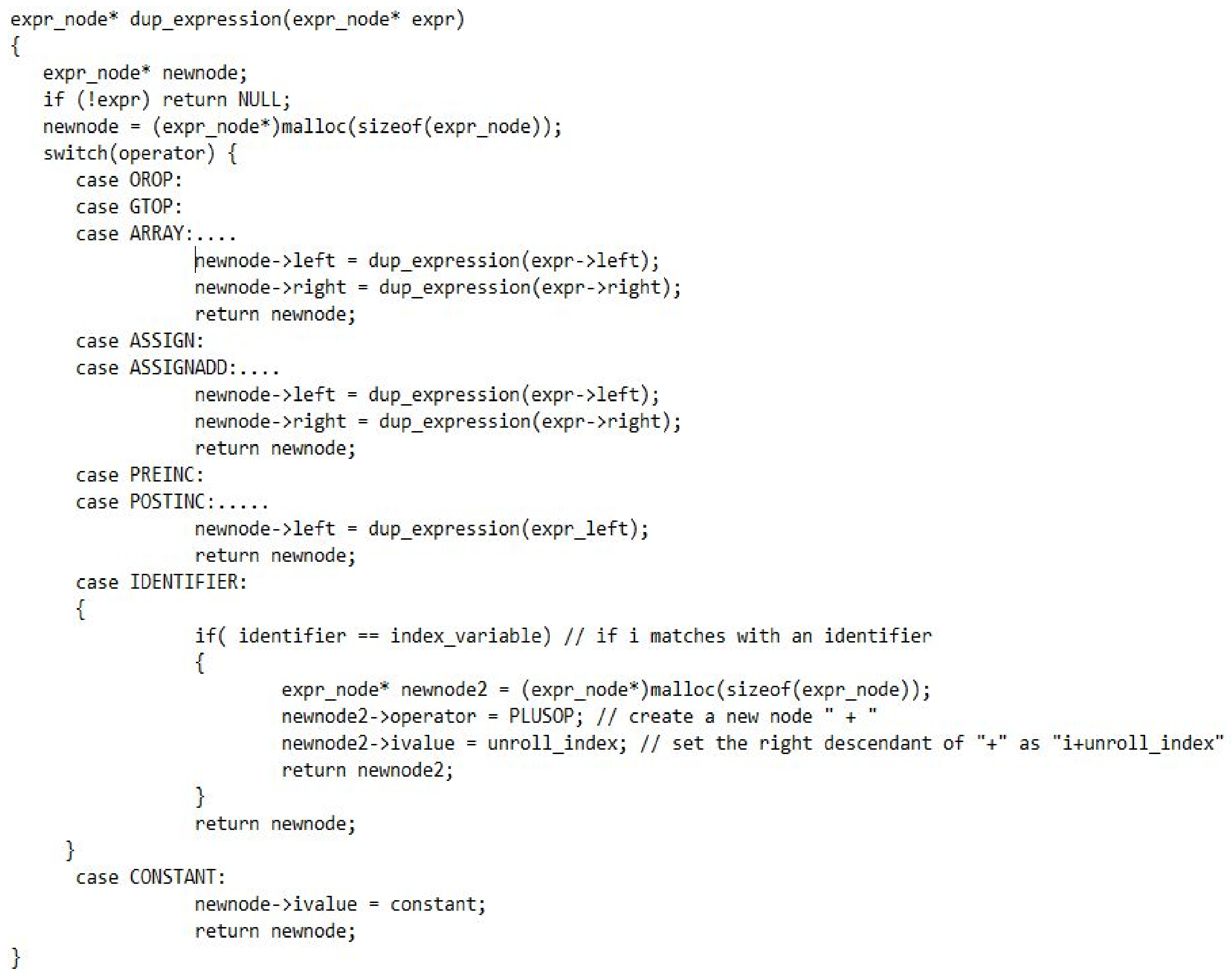
Let us give an example for the above operation. The first code illustrates the initial header of the loop, while the second shows the transformed:



```
for(i=0; i<N; i++){

...

}


for(i=0; i<(N-remainder); i+=unroll_factor){

...

}
```

We develop the process of automatic loop unrolling by creating the void function; unroll. The operation of this function is to produce duplicates for each instruction of the loop body and place them at the end of the loop body. More specifically, we have built an interconnected list which starts from the end of the initial loop body and creates instr_nodes in order to expand the body by copying the same instructions with the same order. The number of how many times the unroll function is being called, depends on the unroll_factor. The latter, can be given by user's preference and experiments have shown that this number affects the performance gain. We cannot claim that as the unroll_factor increases, we will get better hardware output. Basically, you can unroll a loop as long as you can put more resources to work and you stop when you no longer can measure any performance gains. Memory issues and register pressure are the most popular problems that you may face. In our experiments with loop unrolling, we found that – as is the case with software – the code explosion caused by loop unrolling can be concern in hardware design as well. This is because the larger number of operations in the design after loop unrolling have to be mapped to the same number of resources as before. This leads to more complex interconnect (multiplexers) and associated control logic. The size of the FSM controller also increases since more states are required to execute the loop body (even though the number of iterations are fewer).

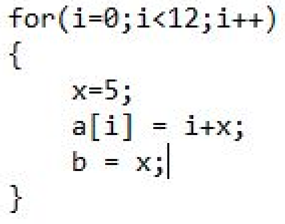
For each expression of every instruction (instr_node), unroll function calls another function; dup_expression whose purpose is to expand the initial loop body by recursively copying the expression of each instruction. More specifically, depending on the operator of each expression, this function recursively builds an abstract syntax tree. We have included every C language operator, so as to cover all the possible cases. In every case, we call again dup_expression setting as parameters the left or the right descendant. The idea is to reach an identifier or a constant case, so as to create a leaf. The pseudocode is being presented in Figure 10, below, and we have to clarify that the unroll_index is the variable that increments by one, on each unrolled loop.



```
expr_node* dup_expression(expr_node* expr)
{
   expr_node* newnode;
   if (!expr) return NULL;
   newnode = (expr_node*)malloc(sizeof(expr_node));
   switch(operator) {
      case OROP:
      case GTOP:
      case ARRAY:....
                newnode->left = dup_expression(expr->left);
                newnode->right = dup_expression(expr->right);
                return newnode;
      case ASSIGN:
      case ASSIGNADD:....
                newnode->left = dup_expression(expr->left);
                newnode->right = dup_expression(expr->right);
                return newnode;
      case PREINC:
      case POSTINC:.....
                newnode->left = dup_expression(expr_left);
                return newnode;
      case IDENTIFIER:
      {
                if( identifier == index_variable) // if i matches with an identifier
                {
                        expr_node* newnode2 = (expr_node*)malloc(sizeof(expr_node));
                        newnode2->operator = PLUSOP; // create a new node " + "
                        newnode2->ivalue = unroll_index; // set the right descendant of "+" as "i+unroll_index"
                        return newnode2;
                }
                return newnode;
     }
      case CONSTANT:
                newnode->ivalue = constant;
                return newnode;
}
```

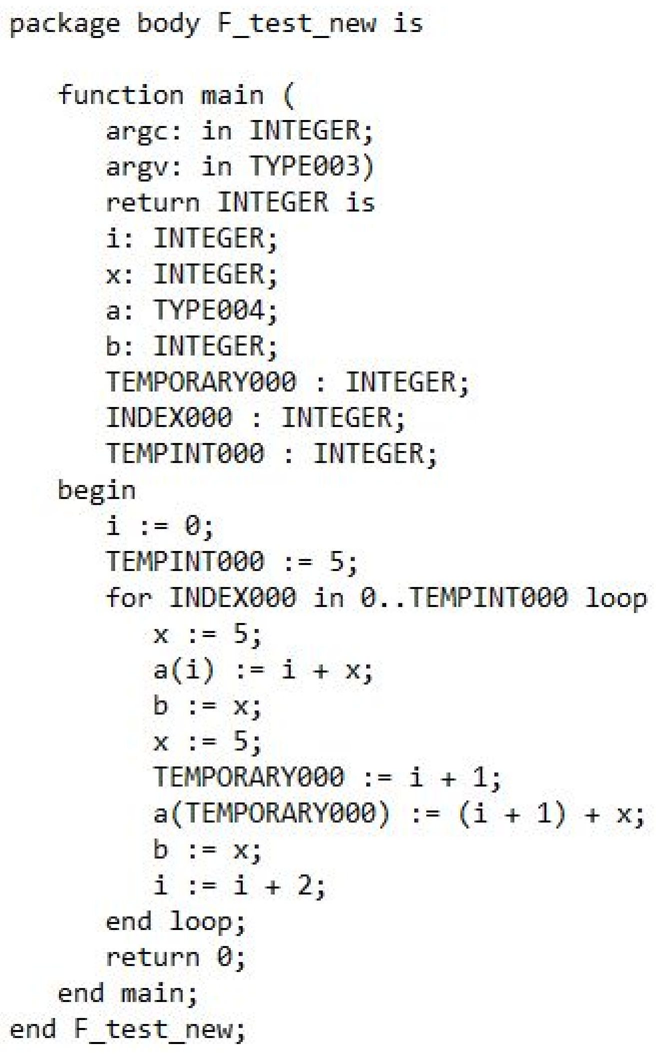
Figure 10. dup_expression pseudocode.

Let us give an example of a loop in which we implement loop unrolling. The code below is a C language input to the ADA translator. In particular, it contains a loop with start limit of twelve iterations and three simple loop instructions. In this example we have unrolled the loop twice, there will be no remainder.



```
for(i=0;i<12;i++)
{
    x=5;
    a[i] = i+x;
    b = x;
}
```

The output ADA code proves our work, because it copies the body of the loop, and each time the index variable is being detected, is incremented by the unroll_index (by one in each iteration). Finally, the step of the loop is also increased by the unroll_factor (by two in this case). Figure 11 illustrates the optimized ADA output code.



```
package body F_test_new is

   function main (
      argc: in INTEGER;
      argv: in TYPE003)
      return INTEGER is
      i: INTEGER;
      x: INTEGER;
      a: TYPE004;
      b: INTEGER;
      TEMPORARY000 : INTEGER;
      INDEX000 : INTEGER;
      TEMPINT000 : INTEGER;
   begin
      i := 0;
      TEMPINT000 := 5;
      for INDEX000 in 0..TEMPINT000 loop
         x := 5;
         a(i) := i + x;
         b := x;
         x := 5;
         TEMPORARY000 := i + 1;
         a(TEMPORARY000) := (i + 1) + x;
         b := x;
         i := i + 2;
      end loop;
      return 0;
   end main;
end F_test_new;
```

Figure 11. Optimized ADA code.

As we have mentioned above, the division between the start limit N of the loop with the unroll factor may give us a remainder. So, in this case we have to create a new FOR_LOOP, that will execute the remaining iterations. The index variable of the new FOR_LOOP must maintain its previous value, i.e. the last value of the execution of the previous loop. Therefore, the first expression of the header must be NULL. In addition, the limit of the new FOR_LOOP, i.e. the second expression of the header, will be the start limit N. Finally, a POSTINC node will be entered as third expression. The body of the new FOR_LOOP is the same as the initial body of the old FOR_LOOP, before the unrolling. As a result, we call the copy function (similar with unroll), in order to duplicate all the instructions. As a consequence, we have to call again dup_expression so as to get the corresponding expressions of these instructions.
In the pseudocode below we show a process of creating these nodes.

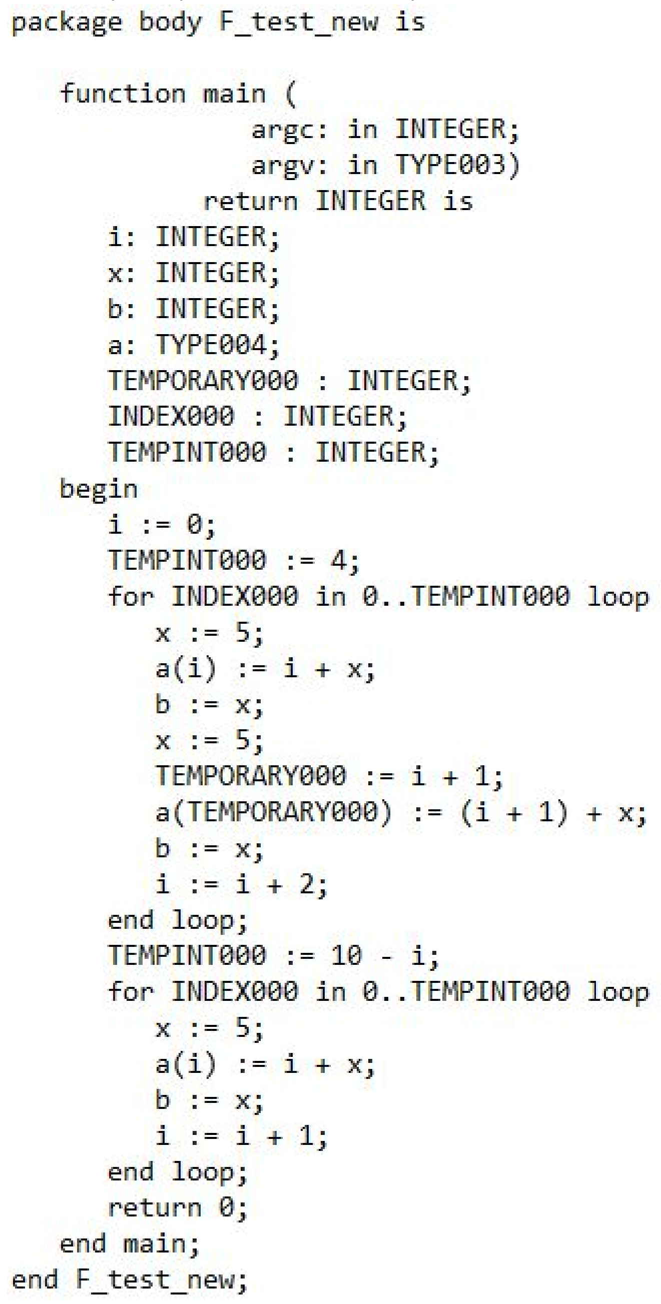
```
if(remainder!=0)
{

    node_for->type = FOR_LOOP;
    node_for->first_expression = NULL;
    node_for->second_expression->operator = {LTOP,GTOP..};
    node_for->second_expression->left = index_variable; // step i
    node_for->second_expression->right = N;
    node_for->third_expression->operator = POSTINC;
    node_for->third_expression->left = index_variable;
    node_for->tail_instruction = copy();
}
```

In the previous example, if we change the limit from twelve to eleven in the C language input, we will have a remainder of one iteration. The unrolled FOR_LOOP body will remain the same, but the only addition would be the new FOR_LOOP related with the remaining iteration. The new loop is being presented below and Figure 13 shows the ADA output for the rest iteration.

```
for(; i<11; i++)
{
        x=5;
        a[i] = i+x;
        b = x;
}
```

As you can observe in Figure 12, the new FOR_LOOP that we have created it is being executed correctly, only once.

In conclusion, the final step of our work is to make experiments using as benchmarks the same Mpeg2 codes that have been used in Section 4. In particular, only recon.c and spatscal.c have suitable loops for testing our loop unrolling optimization. We will present the comparison (in FSM states) of the PARCS version before and after implementing loop unrolling. Table III summarizes the results of automatic-dynamic loop unrolling in the way they have been analyzed in Table II.



```
package body F_test_new is

   function main (
               argc: in INTEGER;
               argv: in TYPE003)
            return INTEGER is
      i: INTEGER;
      x: INTEGER;
      b: INTEGER;
      a: TYPE004;
      TEMPORARY000 : INTEGER;
      INDEX000 : INTEGER;
      TEMPINT000 : INTEGER;
   begin
      i := 0;
      TEMPINT000 := 4;
      for INDEX000 in 0..TEMPINT000 loop
         x := 5;
         a(i) := i + x;
         b := x;
         x := 5;
         TEMPORARY000 := i + 1;
         a(TEMPORARY000) := (i + 1) + x;
         b := x;
         i := i + 2;
      end loop;
      TEMPINT000 := 10 - i;
      for INDEX000 in 0..TEMPINT000 loop
         x := 5;
         a(i) := i + x;
         b := x;
         i := i + 1;
      end loop;
      return 0;
   end main;
end F_test_new;
```

Figure 12. Optimized ADA with remainder.

Table III not only does justify our work because we have profit in each comparison, but also we can figure out that as the unroll_factor increases, we gain better hardware performance. Especially, the comparisons between recon vs recon_4k (4 × unroll) and spatscal vs spatscal_4k (4× unroll) gave us obviously better results than comparing recon vs recon_2k (2× unroll) and spatscal vs spatscal_2k (2× unroll).

File name Comparison (.c)	**Mk / (k*M)**	**States Profit (k*M - Mk)**
recon vs recon_2k	619 / (2*362) = 0.85	105
recon vs recon_4k	939/(4*362) = 0.65	509
recon_2k vs recon_4k	939/(619*2) = 0.76	299
spatscal vs spatscal_2k	489/(2*370) = 0.66	251
spatscal vs spatscal_4k	618/(4*370) = 0.42	862
spatscal_2k vs spatscal_4k	618/(2*489) = 0.50	309

Table III. States profit when comparing results of automatic loop unrolling.

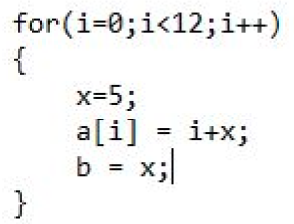
Furthermore, we can advocate that the effectiveness of loop unrolling is closely related with the size of a loop body. For instance, the number of instructions in spatscal loop bodies is by far greater than the corresponding number of recon loop bodies.

6. Conclusions and Future Work

In this thesis work, we have presented valuable optimizations in the source level of the front end of the CCC high-level synthesis tool, such as loop unrolling, function inlining and code motion that implemented firstly by hand. Subsequently, because of the encouraging results that we have obtained, we tried to embed the most significant of these optimizations (loop unrolling) into the front end of the tool. So the main purpose of our thesis, was to successfully apply automatic loop unrolling in order to increase performance and provide more efficient hardware output.

Comparing the results of Table III with Table II (only the unrolled files), the state profit in Table II is better than the corresponding in Table III. There are two reasons that explain this conclusion, firstly the optimizations that we did by hand, do not include remaining iterations (that we get, if there is a remainder), so the state number is rationally less than the corresponding number on automatic loop unrolling process. Secondly, the optimizations that we did by hand in recon and spatscal loops include variables renaming that avoid RAW and WAW dependencies. Moreover, lines 2,3 and 4 in Table II compare loop unrolling optimization with loop unrolling enhanced with code motion. Obviously, loop unrolling with code motion provide less states because of the dependencies elimination and instructions reordering.

Our future work plan is to add code motion technique into the CCC HLS tool in order to automatically provide far better state profit. We have begun this procedure and we have as far achieved to reorder instructions of the unrolled loops. What is left is to rename the reordered instructions in order to achieve greater parallelism. For instance if we use the given example of C language code in section 5:



```
for(i=0;i<12;i++)
{
    x=5;
    a[i] = i+x;
    b = x;
}
```

If we suppose unroll_factor equal to two, we have achieved to do the following:

```
for (i=0; i<12; i+=2)
{
    x = 5;
    x = 5;
    a[i] = i+x;
    a[i+1] = i+1+x;
    b = x;
    b = x;
}
```

So, we intend in our future work to get the fully optimized following code:

```
for (i=0; i<12; i+=2)
{
    x1 = 5;
    x2 = 5;
    a[i] = i+x1;
    a[i+1] = i+1+x2;
    b1 = x1;
    b2 = x2;
}
```

Table IV summarizes the comparisons between *automatic* loop unrolling files with the files that have been transformed by hand and include loop unrolling enhanced with code motion. As we explained above, the latter provide greater states results, but our goal is to *automatically* achieve similar results.

File name Comparison (.c)	**(Mk / M)**	**States Profit (M - Mk)**
recon_2k vs recon_2k_1 (with code motion)	428 / 619 = 0.69	191
recon_4k vs recon_4k_1 (with code motion)	602/ 939 = 0.64	337
spatscal_2k vs spatscal_2k_1 (with code motion)	409/618 = 0.66	209

Table IV. States profit when comparing results of automatic loop unrolling with loop unrolling & code motion (transformed by hand).

On the whole, The contribution of the CCC tool is invaluable with the combination of the C front end. In a matter of minutes large, real-life applications such as MPEG2 engine are formally transformed into provably-correct hardware implementations. The PARCS optimizer delivers high-quality HDL code. Source-level optimizations like loop unrolling, function inlining and compile-level code scheduling, coupled with PARCS boost performance of the CCC tool significantly. Further, it is clear that the two tools, the source-level optimizer and PARCS, complement one-another, building an optimizing tool that can optimize hardware at the level of state-of-the-art.

7. References

[1] Gal, B. L., Casseau, E., and Huet, S. Dynamic Memory Access Management for High-Performance DSP Applications Using High-Level Synthesis. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 16, No. 11, pp.1454-1464 , November 2008.

[2] Gupta, S., Gupta, R. K., Dutt, N. D., and Nikolau, A. Coordinated Parallelizing Compiler Optimizations and High-Level Synthesis. In ACM Transactions on Design Automation of Electronic Systems. Vol. 9, No. 4, pp. 441–470 , 2004.

[3] Walker, R. A., and Chaudhuri, S. Introduction to the scheduling problem. In IEEE Design & Test of Computers. Vol. 12, No. 2, pp. 60–69, 1995.

[4] Dossis, M. F. A Formal Design Framework to Generate Coprocessors with Implementation Options. In International Journal of Research and Reviews in Computer Science (IJRRCS, ISSN: 2079-2557). Science Academy Publisher, UnitedKingdom, Vol.2, No.4,pp.929-936, August2011, DOI=http://www.sciacademypublisher.com.

[5] Paulin, P. G., and Knight, J. P. Force-directed scheduling for the behavioral synthesis of ASICs. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 8, No. 6, pp. 661–679, December 1989.

[6] Kountouris, A. A., and Wolinski, C. Efficient Scheduling of Conditional Behaviors for High-Level Synthesis. In ACM Transactions on Design Automation of Electronic Systems. Vol. 7, No. 3, pp. 380–412 , 2002.

[7] Del Barrio, A. A., Hermida, R., Memik, S. O., Mend´ıas, Jos´e M., and Molina, Mar´ıa C. Multispeculative Addition Applied to Datapath Synthesis. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 31, No. 12, pp. 1817-1830, December 2012.

[8] Sarbishei, O., and Radecka, K. On the Fixed-Point Accuracy Analysis and Optimization of Polynomial Specifications. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 32, No. 6, pp. 831-844, June 2013.

[9] Morvan, A., Derrien, S., and Quinton, P. Polyhedral Bubble Insertion: A Method to Improve Nested Loop Pipelining for High-Level Synthesis. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 32, No. 3, pp. 339-352, March 2013.

[10] Banerjee, K., Karfa, C., Sarkar, D., and Mandal, C. Verification of Code Motion Techniques Using Value Propagation. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 33, No. 8, pp. 1180-1193, August 2014.

[11] Sierra, R., Carreras, C., Caffarena, G., and López Barrio, C. A. A Formal Method for Optimal High-Level Casting of Heterogeneous Fixed-Point Adders and Subtractors. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 34, No. 1, pp. 52-62, January 2015.

[12] Xydis, S., Palermo, G., Zaccaria, V., and Silvano, C. SPIRIT: Spectral-Aware Pareto Iterative Refinement Optimization for Supervised High-Level Synthesis. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 34, No. 1, pp. 155-159, January 2015.

[13] Gupta, S., Dutt, N., Gupta, R., and Nicolau, A. Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow. Center for Embedded Computer Systems Technical Report #03-14, University of California at Irvine, April 2003.

[14] Plesco, A., and Risset, T. Coupling Loop Transformations and High-Level Synthesis. In RenPar'18 / SympA'2008 / CFSE'6, Fribourg, Switzerland, February 2008.

[15] Cardoso, J., Diniz P., and Weinhardt, M. Compiling for Reconfigurable Computing: A Survey. In ACM Computing Surveys, Vol. 42, No. 4, June 2010.

[16] Dimitriou, G., and Dossis,M. Experimenting with a High-Level Synthesis System Front End. In PACET 2015, Ioannina, May 2015; also in Journal of Engineering Science and Technology Review 4 (1) (2013) pp. 68-73.

[17] Dossis, M. High-Level Synthesis and Practical Issues. In PACET 2015, Ioannina, May 2015.