apple core

SEVENTH FRAMEWORK PROGRAMME

FP7-215216

Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs

Specific Targeted Research Project (STReP) THEME ICT-1-3.4

Report on 1st Version of $C2\mu TC/SL$ Source to Source Compiler

Deliverable D3.3, Issue 1.5

Workpackage WP3

Author(s):	George Manis		
Reviewer(s):	Chris Jesshope		
WP/Task No.:	WP3	**Number of pages:**	25
Issue date:	2010/09/30	**Dissemination level:**	Public

Purpose: Describe the 1st version of $C2\mu TC/SL$ compiler.
Results: A detailed description of compiler, implementation details and evaluation results.
Conclusion: $C2\mu TC/SL$ can extract sufficient parallelism from C programs and produce reliable code targeting the SVP parallel programming model.

Approved by the project coordinator: Yes **Date of delivery to the EC:** 2010/09/30

Document history

When	Who	Comments
2010/08/18	George Manis	v1.0 – Initial version
2010/08/19	Chris Jesshope	Comments on v1.0
2010/08/29	George Manis	v1.1 – Modifications on sections 5 and 7.2
2010/09/07	Chris Jesshope	Comments on v1.1
2010/09/14	George Manis	v1.2 – Minor corrections
2010/09/17	Chris Jesshope	Comments on v1.2
2010/09/24	George Manis	v1.3 – New experimental results for different numbers of cores
2010/09/25	Chris Jesshope	Comments on v1.3
2010/09/29	George Manis	v1.4 – New experimental results, mini-manual revised
2010/09/30	Chris Jesshope	Comments on v1.4
2010/09/30	George Manis	v1.5 – Minor corrections, some figures replaced
2010/09/30	Chris Jesshope	Document approved

SEVENTH FRAMEWORK PROGRAMME

Project co-funded by the European Commission within the 7^{th} Framework Programme (2007-11).

Table of Contents

1 Introduction

This document aims to present the 1st version of the $C2\mu TC/SL$ source to source compiler. More specifically, it will present the transformations supported so far, implementation details and a evaluation section. The source code of the 1st version is publicly available.

The implementation has been done using the CoSy[1] [1] compiler development system. The source language is the C programming language and the target is SL, a script language on top of μTC which was our initial target.

This deliverable is part of the AppleCore project (Architecture Paradigms and Programming Languages for Efficient programming of multiple CORES), funded by the Commission of European Communities under the 7th RTD Framework Programme and the Grant Agreement No 215216.

2 The SVP Model, the μTC and the SL Languages

For completeness we give a short description of the SVP [2, 3] model, and the two languages related to this model: SL and μTC [4].

SVP is a model concurrent processor, which supports self-adaptiveness and supplies a set of concurrent instructions. Computation in SVP is considered as a packet of information, which identifies a place, somewhere either locally or remotely where the concurrent program will execute. The communication, if required, comprises data and some definition of its functionality. This packet defines a unit of work, which is an invocation from a thread of all its subordinate threads at a place chosen for its execution. In SVP, a unit of work is represented as a parameterized family of threads that has collective properties defining its computation and mission goals. Because these threads will be required to capture simple hardware semantics, the thread in this model is blocking, which means that it will only proceed, provided that its operations have data to consume.

The definition of a thread specifies the data inputs and outputs of its family, and family termination defines a synchronization point with respect to the family's outputs. Within a family, threads may communicate and synchronize with each other on scalar items of data, allowing the family to encapsulate regularity and locality. All operations must synchronize on their operands and a thread locks until an operand is available. This is important in a concurrently executing model as operands may come from other threads or data may be distributed.

To better describe the SVP model, an intermediate language has been defined. It is called *microthreaded C* (μTC) and it is similar to other concurrent languages based on C. However, no other language implements concurrency in such a dynamic way, using the concept of families of threads with preemption. An advantage of μTC over other approaches is that it allows an abstract representation of maximal concurrency in a schedule–independent form. μTC is a profound but simple extension to C, allowing to capture the thread-based concurrency.

The construct of μTC used for the definition of families of concurrent microthreads over an index variable is `create`. This construct evaluates its parameters and creates threads in index order and dynamically allocates a context of synchronizing memory to each thread it creates. In order to define the scalar variables that are located in the synchronizing memory, we use the type qualifier `shared`. A *shared* variable can be written by one the threads and read by the following in the index sequence. A thread blocks on a shared variable when the shared variable has not been written by the previous thread. The *index* of the family is defined with the type qualifier `index`. For the detection of the termination of a family the construct `sync` is used. This construct blocks until all threads of the family have completed their execution. The shared memory is synchronized and, thus, can be considered as consistent only after `sync` is executed.

SL does not provide more functionality than μTC. SL is a script language on top of μTC which masks some details and provide a more user friendly interface to the programmer. SL uses the `sl_def` to define a family, `sl_create` to create a family of threads, the call `sl_sharg` to define a

[1] CoSy® is a registered trademark of ACE Associated Computer Experts bv, Amsterdam, The Netherlands.

shared variable, `sl_glarg` to define a global argument, the calls `sl_getp` and `sl_setp` to read a shared variable and to write a shared variable respectively.

3 The Tool Chain

The compilation procedure for producing binaries from the C input file is shown in figure 1. The $C2\mu TC/SL$ source to source compiler which is the subject of this document and is developed under workpackage III. $C2\mu TC/SL$ takes as input sequential programs written in ANSI C, extracts concurrency and produces source code in SL. SL code is again compiled to produce binaries. Technically the SL compiler produces C code which goes through the standard C compiler but with embedded assembly that provides the concurrency controls. Those binaries can execute on a simulation environment (or on an FPGA prototype at the end of the project).

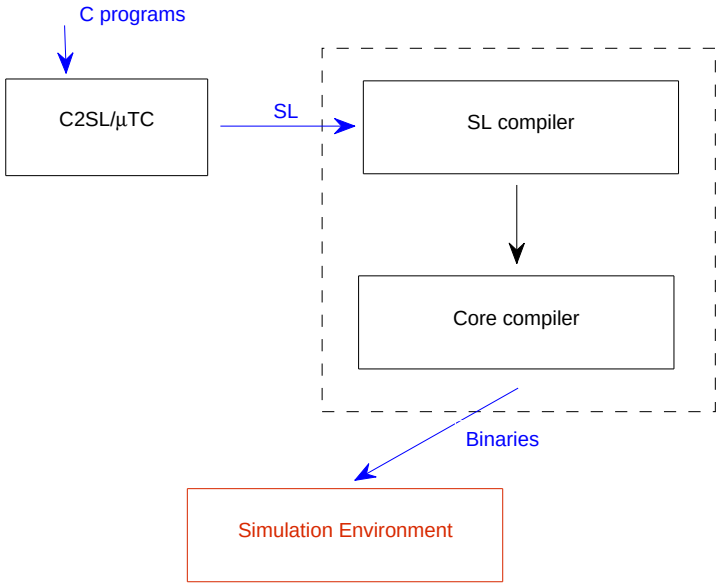


Figure 1: From a C program to the program output

A mini manual on how to produce output, starting from an ANSI C program follows. Suppose we want to compile and execute the program `test.c`. First we analyze the file `test.c` with CoSy:

```
fromCoSy test.c
```

where `fromCoSy` is a compiler we produced using CoSy, extracting information from the source file about which loops exist and which basic blocks they occupy. This compiler also gives us an intermediate representation to work with.

In the next step of the compilation process, we pass the output information of the CoSy compiler to our $C2\mu TC/SL$ compiler (Notice that file extension should not be used):

```
c2sl test
```

If we receive no error messages then a file with the name the prefix `sl_` into the original filename (in our case `sl_test.c`) is produced.

In this phase we need to compile and run the transformed program with the SL compiler and SVP simulator accordingly like this:

```
slc sl_test.c
```

whrer `slc` is the name of the compiler which can take a series of parameters with the most important ones being:

`-o Output` : Instead of `a.out` the output filename is set to `Output`

`-b profile`: `Slc` can compile for a series of profiles but we used the `mta` profile which is the simulator profile. If we do not use the `-b` the compiler generates a sequential version of the program useful for debugging purposes.

Back in our example, we type:

```
slc sl_test.c -b mta
```

And get the `a.out` which is ready to execute on the simulator. The simulator is invoked by the program `slr`. In its basic form `slr` just takes the file as input and nothing else. This will execute the program with a default profile of 1 core. If we need to run on a different number of cores then we use the argument `-n` followed by the number of cores e.g.

```
slr a.out -n 8
```

4 Mapping Loops on SVP

In this section we will briefly present the loop transformations currently supported by $C2\mu TC/SL$. The compiler can extract parallelism from various loops and map them onto the underlying multi–core architecture.

4.1 Simple Loops

Suppose the following loop in which there is one single unary dependency between the elements of an array:

```
for (i=1;i<N;i++) {
   ...
   w[i]=w[i-1]+1;
   ...
   }
```

A family of N threads is created for this loop. Each thread is responsible for one of the iterations. All threads start at the same time and execute in parallel until they reach the dependency. To minimize the delay introduced by the dependency, the synchronizing memory will be used and a shared variable will be created.

If we have a single dependency, but not a unary one then the loop is transformed from a single dimensional loop into a loop of a higher dimension. It is equivalent with x independent loops, where x is the dependency distance, the first one starting from x the second starting from $x+1$ and the last from $2x-1$. If we have more than one dependencies and given that the i_{th} iteration can receive data only from the $(i-1)_{th}$ iteration, the data produced in iterations $i-2, i-3, \ldots, i-x$ have to be transferred to the i_{th} iteration through the iteration $(i-1)$.

4.2 Nested Loops

The transformations of nested loops are not just a straightforward generalization of those of simple loops. We will not study here the simpler cases which are generalizations of simple loops.

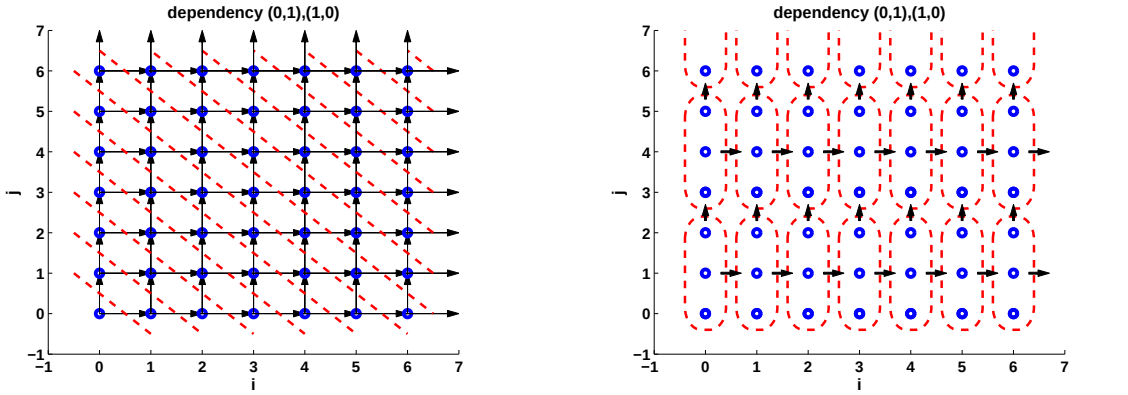
In the following example we need to handle two dependencies ($a > 0,\ b > 0,\ c > 0,\ d > 0$).

```
for (i=max(a,c); i<N-1; i++)
  for (j=max(b,d);j<N-1;j++)
    w[i][j]=w[i-a][j-b]+w[i-c][j-d]
```

The processor does not provide support for such kind of communication between iterations. We can handle the problem like most parallelizing compilers do, based on the theory of hyperplane [5], but we look for something better, a customized solution for our processor. Let us consider the following example:

```
for (i=0;i<=N;i++)
    for (j=0;j<=N;j++)
        A[i][j]=A[i-1][j]+A[i][j-1];
```

The dependencies are (0,1) and (1,0). The problem here is that each computation of $A_{i,j}$ requires that both $A_{i,j-1}$ and $A_{i-1,j}$ have already been computed. It seems that this code cannot be parallelized. However, according to the hyperplane method [5] we can find which points can be computed in parallel. In figure 2(a) iterations in regions between two dashed lines can run in parallel.



(a) Parallelism in hyperplane

(b) Parallelism using families of threads, here m=3

Figure 2: From the hyperplane to the concurrent families of threads

We would like to customize the hyperplane method for our target architecture. The challenge here is to find a solution which allows more than one family to run in parallel. According to the micro–threaded model, the memory is considered coherent only after the execution of the family is over. Given the two dependencies of our example, two families of threads cannot run in parallel, otherwise data flow between different families of threads is required. However, we can break each one of those families into m smaller ones as shown in figure 2(b). Dashed boxes are families of threads. Upon finishing execution, a family of thread can launch the family on the right hand side and on the top of it as shown by the dark dashed arrow. In this way more than one family can execute in parallel. We exploit both the parallelism resulting from the hyperplane model theory and the underlying hardware, since we have families of threads exploiting the synchronizing memory. The problem is reduced from a problem with dependencies between iterations to a problem with dependencies between groups of successive iterations, i.e. families of threads.

4.3 "While" Loops

The `while` loops are much more difficult to be handled than the `for` loops, at least in the general case. In a while loop the loop variable, the limits of the loop and the step are difficult or impossible to be decided. Generally the programmers prefer to use `while` loops instead of `for` loops when the limits of the loops are decided at run-time.

However, `while` loops are very common and a useful programming tool. In order to support `while` loop structures, we used the `sl_break` command which allows exit from a family and termination of the family when called from inside of a thread. Thus, each `while` loop is transformed to a `while(true)` loop, and when an exit condition is fulfilled one of the threads calls `sl_break`. The question that arise is how to use the `sl_create` structure since `sl_create` creates a number

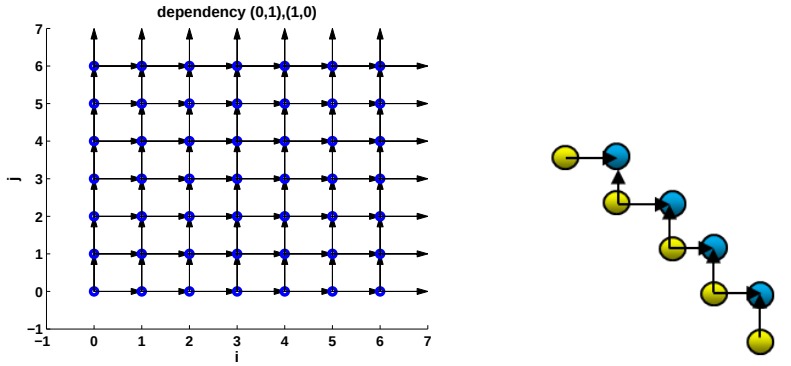
of threads known at the initialization of `sl_create`. The solution is to create a very large number of threads, in blocks, so that first a number of threads equal to the block size are created and if all these threads complete execution successfully, a second block of threads is created, and so on ... This is done in a higher level. Inside a block we exploit a feature of the underlying architecture. The architecture does not create all threads in a block from the beginning and wait for all threads to complete; it creates new threads as threads in the block complete.

5 Run-time Scheduling

Instead of trying to extract parallelism statically during compile-time, we opted to move the scheduling of parallel families of threads to the run-time environment. The information available at run-time is a superset of that available at compile time making decisions more flexible. Please consider the following example code:

```
for (i=1<i<=10;i++)
   for (j=1<j<=10;j++)
      a[i][j]=a[i-1][j]+a[i][j-1]
```

The distance vector is $d = \{(1,0),(0,1)\}$. The index space is shown on figure 3(a).



(a) The index space for $d = \{(0,1),(1,0)\}$

(b) The movement of data

Figure 3: Run-time scheduling example

Once we know in every iteration which threads can run independently (and hence in parallel), then it is straightforward to decide the group of threads that can run in the following iteration. Let us consider one iteration of our example as shown in figure 3(b). The lower diagonal (yellow color) indicates the threads being executed in the current iteration. If we associate an array storing each one of the indices then this array indicates the number of the dependencies been satisfied so far. By increasing the values of this array with the coordinates of the distance vector, we can compute the threads allowed to be executed in the next iteration.

In a multiprocessing system like SVP, we can assign the role of the scheduler onto a thread that executes in parallel with the threads that perform computations. We need to note here that the size of each family is integral to the total throughput of this scheme since the ideal size will spend exactly the same time in computations as the time the scheduler thread needs to calculate the new coordinates. In figure 4 we consider that each family consists of three threads. Again we have two dependencies, (1,0) and (0,1). Green families have completed execution. At this moment the yellow families run. Another thread, a scheduler (marked in figure with yellow color as well), runs in parallel with the yellow threads. Before the yellow families complete execution the scheduler (we consider in the general case in which the size of the family is large enough) decides which are the next families to run and schedules the red families. The cyan families will be scheduled later.

From all the above we can formulate an algorithm that executes at run-time:

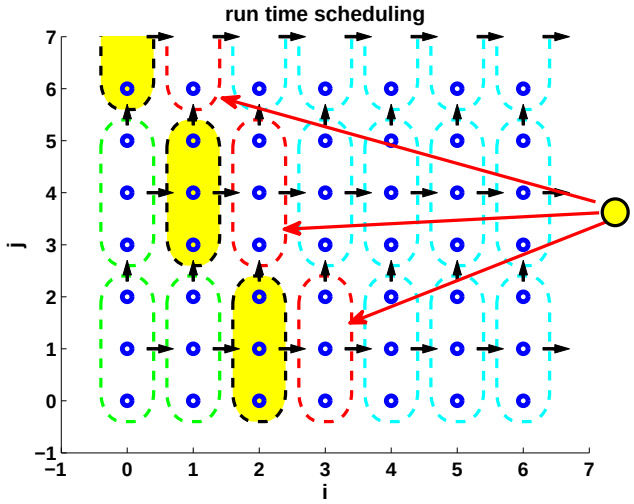


Figure 4: The run-time scheduler.

1. Allocate a multidimensional array, with the same number of dimensions as the nesting level of the loop. The last (innermost) dimension is divided by N. Suppose $N = 3$, the index space breaks into families as shown in figure 2(b)

2. Consider the two dimension vectors v_1 and v_2 that start empty. Initialize each array element with the number of dependencies already satisfied before the computation starts. Add the coordinates of the elements that satisfy the total number of dependencies in vector v_1.

3. Create a family of threads with sizeof(v_1)+1 threads. Perform computation and synchronize the family. If the vector v_2 is empty then jump to the end of computation. Copy the contents of v_2 to v_1. Clear v_2. Jump to the third step.

By *perform computation* we mean:

```
Perform_Computation():
  index i;
  if i<length(v1)+1 then
    create family of N threads
      Loop_body
    sync_family
    return
  else
    for each element v in v1
      for each dependence d
        add the values of d to the coordinates v
          increase the array element
            indicated by the new coordinates by 1
          if new array element satisfies
            the total number of dependencies then
              add new coordinates to vector v2.
  end if
```

And thus the entire computation index grid is covered, with as many parallel families running in each iteration as possible. It is worth to notice that we calculate the nodes in the exact same way as the wavefront/hyperplane method but the entire computation now takes place during run-time and thus is able to expose a greater degree of parallelism than the wavefront/hyperplane model as well as provide support for different kinds of dependencies (non-static ones) resulting into higher versatility supported by the mechanisms of the SVP model.

6 Implementation

6.1 Implementation Details

The compiler relies heavily on the CoSy compiler system. CoSy is used to analyze the source code and

- generate the intermediate representation
- retrieve information related to the loop structures and
- all variables present in the code

The loop analysis offers us information on:

- which basic blocks are included in the loop
- which is the loop variable
- which are the bounds of the loop
- the step of the loop variable

Of course, it is not always possible to extract the above information. CoSy returns to us three kinds of basic blocks related to a loop structure:

- the initialization block
- the test block and
- the increment block.

The first one contains information about the initialization of the loop, the second one about the exit condition and the third one on how the loop variable changes between successive loop iterations.

The compiler's basic unit of organization is what we call a *masterloop*. It represents either one loop or multiple loops if they are perfectly nested. To understand whether a loop is perfectly nested or not, we count the number of the basic blocks it contains. In a perfectly nested situation, one loop has three more basic blocks than its nested one (the initialization block, the test block and the increment block). Masterloops can be contained into other masterloops as well. Each masterloop is analyzed for the discovery of dependencies and shared variables.

A variable is deemed to be *shared* if this variable is read before written inside the loop body. However, since only the shared variables keep their modified values after the termination of a family, it is also necessary to check if this variable is read again after the loop body and before the next write access on this variable. In that case the variable is deemed to be a shared again.

The dependence analysis is rudimentary and looks for expressions in the form:

```
Array[index+constant1] = Array [index+constant2] + ...
```

If the expression inside the brackets is more complex, then the loop is deemed to be non-transformable.

Both shared variable analysis and dependence analysis rely on a dependence graph created and keeping count on which variable accesses which one. By traversing this graph we get all the necessary information, for example, if a variable is referenced or not, if a variable has been written before been read or if an array is referenced through constants or not, e.t.c.

Before the output file is created, the masterloops are classified into transformable and non-transformable, In the latter case, loops are further classified into subcategories. A masterloop is non-transformable in the following cases:

- the loop variable or the bounds of the loop or the step can not be decided
- the dependence analysis failed to detect the dependencies

On the rest of the cases the loop is deemed to be transformable. Non-transformable loops are further categorized into:

- loops with dependencies forming a dependence vector
- loops without dependencies and
- loops containing shared variables, meaning that the loop is essentially serially executed

In the output file (SL code) the type declarations is a series of typedefs. Since SL is not able to handle any sort of complex variable declaration (like a multidimensional array for example) the typedefs assign names to all variable types that are needed in the program. After these typedefs and for every masterloop a thread function is created. If the loop is transformable then it is given a name starting with `_inner`, otherwise with `_umloop`, following in both cases by the number of the masterloop, in order to give each loop a unique id and differentiate loops of different functions.

The arguments inside the thread definition are named with the prefix `_` in front of the name they had in the C program. Inside the thread function body, the real values of the arguments are passed into local variables. These variables keep the original argument names (the one they had in the C program). The SL call used for passing the arguments is `sl_getp()`.

In the beginning of each thread the index variable is declared through the SL call `sl_index()`.

The code of the loop body follows inside the thread function. It is actually the intermediate representation returned by the CoSy's analysis, reverse-engineered in order to produce again C (or more precisely SL) code. Each basic block is declared with a different label. The control flow remains the same as returned by the CoSy's intermediate representation and uses "gotos" to labels. Extra care needs to be taken when statements make jumps outside of the loop body. These statements are illegal and need to be replaced with the `sl_break()` call which stops the execution of the family.

While converting the basic blocks from CoSy's intermediate representation to SL code, we need also to check if a masterloop is contained inside the loop body. This is achieved by assigning all basic blocks to their respective loop owner and all loops to their masterloop owner. Once a basic block not belonging in the current masterloop is encountered then an `sl_create` call is produced there. Depending on the type of the masterloop, different types of creates are produced. The masterloop is marked properly so that duplicate creations are avoided.

In non-transformable loops that come from while loops, the testing block (that checks the condition of the while) is introduced inside the thread body followed by the `sl_break` command as described in section 4.3.

At the end of the inner umloop thread body, all shared variables are written so that they will be used again by the next thread of the family.

If the masterloop is transformable with a dependence vector then a second utility thread function is created called `_wrapper` (with the number of the masterloop attached in the end) where the discovery of the next cycle takes place. In the last part of the file, the `t_main` thread is created. In the beginning all variables are declared and then code begins to be outputted in the same manner as the `_inner` thread functions. Every time a basic block that belongs to a masterloop is encountered, a specific `create` is produced for that masterloop. The `create` produced depends on the type of the masterloop encountered.

- If the masterloop has at least one shared variable (which signifies a serial execution) then a number of for loops is created (the number of the masterloop dimensionality minus 1) followed by an `sl_create() - sl_sync()` pair of calls which effectively runs the masterloop sequentially while at the same time utilizing the mechanisms provided by the SVP model for shared memory usage

- If the masterloop has no dependencies and no shared variables then a single `sl_create() - sl_sync()` call is produced. This scheme effectively creates all loops in parallel and lets the SVP model handle all scheduling and execution

- If the masterloop contains a dependence vector then our run time scheduling algorithm needs to be implemented inside the source code. So first, the array which contains the entire search space for our scheduler is allocated in memory and initialized with the starting values of how many families of threads can run in parallel. Then a `while(true)` code is generated which

constantly creates the next cycle of families for execution plus the thread which computes the next step. When the next step is empty then the while loop breaks its cycle.

- If we have a masterloop that came from an originally while loop (we identify this by our lack of knowledge of a loop variable or its start, end and step values) then we attempt to use the shared memory of the SVP model to simulate the while execution. For that we create a thread function that contains the loop body (called *umloop*) but inside the loop body we also apply insert as code the checking of the while condition. So, if the condition is met then the umloop will call `sl_break` to stop its execution. As well as that the entirety of the loop is searched for "gotos" that point outside the loop as they also mean an `sl_break` is needed to substitute those "gotos". In the main function, a `while(true)` loop is outputted which contains a `create - sync` pair for the umloop function. The number of threads to be created used there is a macro which can be defined. After the `sync`, a check is done to see whether or not the umloop finished properly or a break was issued. To know that, we create a small array of one dimension which corresponds to all masterloops. Before an `sl_break` command is issued we change the corresponding value of the array to 1 and thus in the main function we can check whether or not the loop exited naturally or after a break. So, if the array has a value of 1 then we break from the `while(true)` loop otherwise we keep running it.

- A small variation of the above is that if the masterloop cannot be transformed yet we do know the start, end, step values of it then we don't need the `while(true) - break` mechanism described above and a single `create` is enough.

- If the masterloop has been deemed non-transformable from the dependency analyzer then no change takes place and the whole loop is transformed as a series of basic blocks that "goto" back to the start or outside the block.

Regardless of type, if the create has any shared variables involved then all of these variables are read after the synchronization by using the `sl_geta()` call of the SL language.

6.2 Illustrative Examples

6.2.1 Matrix Multiplication

Let us first consider the classic example of matrix multiplication:

```
for (i=0;i<N;i++)
    for (j=0;j<N;j++)
    {
        sum=0;
        for (k=0;k<N;k++)
            sum+=a[i][k]*b[k][j];
        c[i][j]=sum;
    }
```

There are 2 masterloops in this situation. The first one is the inner `for(k)` loop and the second is the perfectly nested `for(i,j)`. Also we can see that the `for(k)` loop needs to run sequentially with `sum` being a shared variable. The nested loop `for(i,j)` can run completely in parallel, each occurrence of a combination (i, j) running the `for(k)` loop.

After transformed, the above example becomes as follows. First let us see the inner function of the 1st (0) masterloop.

```
sl_def(mloop_0_inner, void ,
    sl_shparm(int4 ,_sum),
    sl_glparm(arr8* , _a),
    sl_glparm(int4 , _i),
    sl_glparm(arr8* ,_b),
```

```
    sl_glparm(int4 , _j))
    {
        sl_index(k);
        int4    sum = sl_getp( _sum );
        arr8*    a = sl_getp( _a );
        int4    i = sl_getp( _i );
        arr8*    b = sl_getp( _b );
        int4    j = sl_getp( _j );
bb6:;
        sum = sum + (a[i][k] * b[k][j]) ;
bb7:;
    sl_setp(_sum, sum);
} sl_enddef
```

Then the inner function of the 2nd (1) masterloop.

```
sl_def(mloop_1_inner, void,
    sl_glparm ( int4 , _sum),
    sl_glparm ( int4 , _k ),
    sl_glparm ( int4 , _N ),
    sl_glparm ( arr8* , _a ),
    sl_glparm ( arr8* , _b ),
    sl_glparm ( arr8* , _c ),
    sl_glparm(int4 , _i))
    {
        sl_index( j );
        int4    sum = sl_getp(_sum);
        int4    k = sl_getp(_k);
        int4    N = sl_getp(_N);
        arr8*    a = sl_getp(_a);
        int4    i = sl_getp(_i);
        arr8*    b = sl_getp(_b);
        arr8*    c = sl_getp(_c);
bb4:;
        sum  = 0;
        k  = 0;
    {
        sl_create( ,,0, N, 1,,,mloop_0_inner,
            sl_sharg(int4 , _sum, sum) ,
            sl_glarg(arr8* , _a, a) ,
            sl_glarg(int4 , _i, i) ,
            sl_glarg(arr8* , _b, b) ,
            sl_glarg(int4 , _j, j));
        sl_sync();

        sum = sl_geta(_sum);
   }
} sl_enddef
```

Since i and j execute in parallel, another thread function appears to call the `inner_1` function in parallel for the iterator j

```
sl_def(mloop_1_j,void,
    sl_glparm(int4 , _sum),
    sl_glparm(int4 , _k),
    sl_glparm(int4 , _N),
    sl_glparm(arr8* , _a),
    sl_glparm(arr8* , _b),
    sl_glparm(arr8* , _c) )
```

```
    {
        sl_index(i);
        int4    sum = sl_getp(_sum);
        int4    k = sl_getp(_k);
        int4    N = sl_getp(_N);
        arr8*    a = sl_getp(_a);
        arr8*    b = sl_getp(_b);
        arr8*    c = sl_getp(_c);

    sl_create( ,,0, N, 1,,,mloop_1_inner,
        sl_glarg(int4 , _sum, sum),
        sl_glarg(int4 , _k, k),
        sl_glarg(int4 , _N, N),
        sl_glarg(arr8* , _a, a),
        sl_glarg(arr8* , _b, b),
        sl_glarg(arr8* , _c, c),
        sl_glarg(int4 , _i, i) );
    sl_sync( );
} sl_enddef
```

Finally the main function that calls the `mloop_1_j` function for the iterator i.

```
sl_def(t_main,void) {
    int   i;
    int   j;
    int   k;
    int   N;
    int   a[30][30];
    int   b[30][30];
    int   c[30][30];
    int   sum;
    int   l;
bb0:;
    N  = 30;
    i  = 0;
    //goto bb1;
//base:1 - true
    {
        sl_create(,,0, N, 1,,,mloop_1_j,
            sl_glarg(int4 , _sum, sum),
            sl_glarg(int4 , _k, k),
            sl_glarg(int4 , _N, N),
            sl_glarg(arr8* , _a, a),
            sl_glarg(arr8* , _b, b),
            sl_glarg(arr8* , _c, c));
        sl_sync( );
    }
bb12:; bb13:;
}
sl_enddef
```

6.2.2 Selection Sort

Let us consider now the example of the selection sort algorithm given below:

```
    for (i=0;i<10;i++)
    {
        min=a[i];
        pos=i;

        for (j=i+1;j<10;j++)
            if (a[j]<min)
            {
                min=a[j];
                pos=j;
            }

        min=a[i];
        a[i]=a[pos];
        a[pos]=min;
    }
```

We can detect two masterloops in this example, the j and i masterloops. The code that is automatically produced by this algorithm is listed below. First, the j thread function:

```
sl_def(mloop_1_inner, void ,
    sl_glparm(int4* , _a),
    sl_shparm(int4 , _min),
    sl_shparm(int4 , _pos))
    {
        sl_index(j);
        int4*    a = sl_getp( _a );
        int4    min = sl_getp( _min );
        int4    pos = sl_getp( _pos );
bb8:;
        if ( a [ j  ]   < min  ) goto  bb9;  else goto bb10;
bb9:;
        min  = a [ j  ]  ;
        pos  = j ;
bb10:;
bb11:;
    sl_setp(_min, min);
    sl_setp(_pos, pos);
} sl_enddef
```

The *min* variable is considered shared since it is read before it is written in this code and the *pos* variable is considered shared since it is accessed later on after this loop has finished.

The i masterloop:

```
sl_def(umloop_2, void,
    sl_glparm(int4 , _min),
    sl_glparm(int4* ,_a),
    sl_shparm(int4 , _i),
    sl_glparm(int4 , _pos),
    sl_glparm(int4 ,_j))
    {
        int4    min = sl_getp( _min );
        int4*    a = sl_getp( _a );
        int4    i = sl_getp( _i );
        int4    pos = sl_getp( _pos );
```

```
        int4    j = sl_getp( _j );
bb5:;
    if ( i  < 10 ) goto  bb6;  else _result[2]=1;sl_break();;
bb6:;
    min  = a [ i  ]  ;
    pos  = i ;
    j  = i  + 1;
    {
        sl_create( ,,i + 1, 10, 1,,,mloop_1_inner,
            sl_glarg(int4* , _a, a) ,
            sl_sharg(int4 , _min, min) ,
            sl_sharg(int4 , _pos, pos));
        sl_sync();

        min = sl_geta(_min);
        pos = sl_geta(_pos);
    }
bb12:;
    min  = a [ i  ]  ;
    a [ i  ]   = a [ pos  ]  ;
    a [ pos  ]   = min ;
bb13:;
    i  = i  + 1;
    sl_setp(_i, i);
} sl_enddef
```

The masterloop was considered as untransformable with i as a shared variable to ensure sequential execution. In the main function this is the code that handles the call of the thread i function:

```
        sl_create( ,,0, 10, 1,,,umloop_2,
            sl_glarg(int4 , _min, min) ,
            sl_glarg(int4* , _a, a) ,
            sl_sharg(int4 , _i, i) ,
            sl_glarg(int4 , _pos, pos) ,
            sl_glarg(int4 , _j, j));
        sl_sync();
        i = sl_geta(_i);
```

7 Evaluation

In this section we will present some evaluation results. In the first part we evaluate the $C2\mu TC/SL$ with respect to its ability to automatically extract parallelism from loop structures. In the first part we chose not to present experimental results and figures but a qualitative evaluation which could express better the ability of the compiler to capture parallelism. As input, a well accepted, widely used and customized for loop structures suite of benchmarks was used: the Livermore Loops. In the second part, experimental results are presented, using the whole tool chain as described in section 3.

7.1 Parallelism Extraction from the Livermore Loops

The Livermore Loops is a benchmark for parallel computers. It consists of 24 kernels. The benchmark was published in 1986 [6, 7]. Each kernel carries out a different mathematical problem. Those kernels are: hydrodynamics fragment,incomplete Cholesky conjugate gradient, inner product, banded linear systems solution, tridiagonal linear systems solution, general linear recurrence

equations, equation of state fragment, alternating direction implicit integration, integrate predictors, difference predictors, first sum, first difference, 2-D particle in a cell, 1-D particle in a cell, casual Fortran, Monte Carlo search, implicit conditional computation, 2-D explicit hydrodynamics fragment, general linear recurrence equations, discrete ordinates transport, matrix-matrix transport, Planckian distribution, 2-D implicit hydrodynamics fragment, location of a first array minimum.

We selected the Livermore Loops since $C2\mu TC$ focuses on loops. There are many benchmarks suits available which use loops and were candidates for this first qualitative evaluation. Livermore Loops seem to be more appropriate since they are small and can be discussed in detail, the number of kernels is relatively large compared to other suits, they consist of kernels solving different problems from various fields and finally not all of them can be parallelized. The automatic detection or extraction of parallelism is not easy and, thus, constitute a good indicator for our evaluation. In the rest of this section we discuss how our compiler can extract parallelism from the Livermore Loops. We will investigate one by one all kernels and we will evaluate its ability to identify the expected parallelism.

7.1.1 Kernel 1 – hydro fragment

```
for ( k=0 ; k<n ; k++ )
   x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] ) ;
```

The first kernel is a simple nested loop in which the element x_k of array x depends on the element y_k of array y and z_{k+10} and z_{k+11} of array z. This loop can be fully parallelized and this parallelism is extracted correctly by the $C2\mu TC/SL$ compiler.

7.1.2 Kernel 2 – incomplete Cholesky conjugate

```
ii = n;
ipntp = 0;
do {
    ipnt = ipntp;
    ipntp += ii;
    ii /= 2;
    i = ipntp - 1;
    for ( k=ipnt+1 ; k<ipntp ; k=k+2 ) {
        i++;
        x[i] = x[k] - v[k]*x[k-1] - v[k+1]*x[k+1];
        }
    }
while ( ii>0 );
```

This is a more complicated kernel and more difficult to extract parallelism. In the `do` loop, variables *ipnt*,*ipntp*,*ii* and i are written inside the loop something that makes the loop seem to be sequential. The inner loop k can give some limited parallelism depending on the values of i and k. We assume it is difficult for most compilers to extract parallelism here. The $C2\mu TC/SL$ compiler can use the synchronizing memory to speed up the execution of the `do` loop, but can not capture any parallelism in the inner loop since the dependency between i and k is not static.

7.1.3 Kernel 3 – inner product

```
q = 0.0;
for ( k=0 ; k<n ; k++ ) {
    q += z[k]*x[k];
    }
```

Table 1: Summarizing briefly the evaluation results for kernels 1–12

Kernel	*Expected Parallelism*	*Parallelism Extracted by C2μTC/SL*
1	full parallelism	full parallelism
2	do loop: serial execution loop k: limited parallelism	fail due to non–static dependencies
3	loop k: tree parallelism	synchronizing memory
4	loop k: serial execution loop j: serial execution	fail due to non–static relations
5	loop i: serial execution	synchronizing memory
6	loop i: serial execution loop k: serial execution	fail due to non–static dependencies
7	full parallelism	full parallelism
8	full parallelism	fail to detect dependencies
9	full parallelism	full parallelism
10	full parallelism	full parallelism
11	loop k: serial execution	synchronizing memory
12	full parallelism	full parallelism

The inner product is an accumulation to a global variable. The variable q is a shared variable and its value accumulated in every thread. $C2\mu TC/SL$ will detect q as shared and will use the synchronizing memory to speedup the execution. Accumulation to q can also give some parallelism, when adding partial sums in parallel to produce the total sum (binary tree parallelism). This kind of parallelism is not supported by $C2\mu TC/SL$ and is not detected here.

7.1.4 Kernel 4 – banded linear equations

```
m = ( 1001-7 )/2;
    for ( k=6 ; k<1001 ; k=k+m ) {
        lw = k - 6;
        temp = x[k-1];
        for ( j=4 ; j<n ; j=j+5 ) {
            temp -= x[lw]*y[j];
            lw++;
            }
        x[k-1] = y[4]*temp;
    }
```

Loop j is an accumulation to a variable. However, there is an relation between the elements x_{lw} and x_{k-1} which can be both dependency and anti–dependency. The code does not seems to be easily automatically parallelized. $C2\mu TC/SL$ fails to extract parallelism due to this non–static relation between x_{lw} and x_{k-1}.

7.1.5 Kernel 5 – tri-diagonal elimination, below diagonal

```
for ( i=1 ; i<n ; i++ )
    x[i] = z[i]*( y[i] - x[i-1] );
```

This serial loop. The value of x_{i-1} will be received by the previous thread, the values of z_i and y_i will be read from the main memory and the value of x_i will be returned in the main memory. The kernel is successfully transformed.

7.1.6 Kernel 6 – general linear recurrence equations

```
for ( i=1 ; i<n ; i++ )
```

```
for ( k=0 ; k<i ; k++ )
    w[i] += b[k][i] * w[(i-k)-1];
```

The kernel consists of two loops. The first ones creates non–static dependencies between iterations. $C2\mu TC/SL$ cannot parallelize this kernel.

7.1.7 Kernel 7 – equation of state fragment

```
for ( k=0 ; k<n ; k++ )
    x[k] = u[k] + r*( z[k] + r*y[k] ) +
           t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
              t*( u[k+6] + r*( u[k+5] + r*u[k+4])));
```

Even though it seems complicated, this is a not a difficult case. Only x_k is modified and then there is no loop carried dependency at all. Parallelism is extracted as expected.

7.1.8 Kernel 8 – ADI integration

The code of this kernel is much larger and will not be listed here. Even though the kernel is possible to be parallelized, the automatic parallelization seems to be difficult, since $C2\mu TC/SL$ fails to identify the dependencies.

7.1.9 Kernel 9 – integrate predictors

```
for ( i=0 ; i<n ; i++ )
    px[i][0] = dm28*px[i][12] + dm27*px[i][11] +
               dm26*px[i][10] + dm25*px[i][ 9] +
               dm24*px[i][ 8] + dm23*px[i][ 7] +
               dm22*px[i][ 6] + c0*( px[i][ 4] +
               px[i][ 5]) + px[i][ 2];
```

An embarrassingly parallel nested loop. It writes to the first element of each row of an array, by reading elements from the same raw. Full parallelism is detected here.

7.1.10 Kernel 10 – difference predictors

This is another embarrassingly parallel nested loop, successfully detected by $C2\mu TC/SL$. Due to the large size of the code, it is not listed here.

7.1.11 Kernel 11 – first sum

```
x[0] = y[0];
for ( k=1 ; k<n ; k++ )
    x[k] = x[k-1] + y[k];
```

In this kernel the loop has one dependence and the execution has to be sequential. $C2\mu TC/SL$ creates a family of threads for the loop which speeds up the execution using the synchronizing memory.

7.1.12 Kernel 12 – first difference

```
for ( k=0 ; k<n ; k++ )
    x[k] = y[k+1] - y[k];
```

Another embarrassingly parallel nested loop. The parallelism is captured successfully.

7.1.13 Kernel 13 – 2-D PIC (Particle In Cell)

```
for ( ip=0 ; ip<n ; ip++ ) {
    i1 = p[ip][0];
    j1 = p[ip][1];
    i1 &= 64-1;
    j1 &= 64-1;
    p[ip][2] += b[j1][i1];
    p[ip][3] += c[j1][i1];
    p[ip][0] += p[ip][2];
    p[ip][1] += p[ip][3];
    i2 = p[ip][0];
    j2 = p[ip][1];
    i2 = ( i2 & 64-1 ) - 1 ;
    j2 = ( j2 & 64-1 ) - 1 ;
    p[ip][0] += y[i2+32];
    p[ip][1] += z[j2+32];
    i2 += e[i2+32];
    j2 += f[j2+32];
    h[j2][i2] += 1;
}
```

A complicated example in which our compiler fails to understand and extract any sort of meaningful parallelism so the loop is kept without any transformation.

7.1.14 Kernel 14 – 1-D PIC (Particle In Cell)

```
for ( k=0 ; k<n ; k++ ) {
    vx[k] = 0.0;
    xx[k] = 0.0;
    ix[k] = (long) grd[k];
    xi[k] = (double) ix[k];
    ex1[k] = ex[ k - 1 ];
    dex1[k] = dex[ ix[k] - 1 ];
}
for ( k=0 ; k<n ; k++ ) {
    vx[k] = vx[k] + ex1[k] + ( xx[k] - xi[k] )*dex1[k];
    xx[k] = xx[k] + vx[k]  + flx;
    ir[k] = xx[k];
    rx[k] = xx[k] - ir[k];
    ir[k] = ( ir[k] & 2048-1 ) + 1;
    xx[k] = rx[k] + ir[k];
}
for ( k=0 ; k<n ; k++ ) {
    rh[ ir[k]-1 ] += 1.0 - rx[k];
    rh[ ir[k]   ] += rx[k];
}
```

There are three distinct loops in this kernel. The first is understood as completely parallel and treated as such. The second and third are considered non-transformable as no parallelism can be detected and extracted.

7.1.15 Kernel 15 – Casual Fortran. Development version

```
ng = 7;
nz = n;
ar = 0.053;
br = 0.073;
for ( j=1 ; j<ng ; j++ ) {
    for ( k=1 ; k<nz ; k++ ) {
```

```
            if ( (j+1) >= ng ) {
                vy[j][k] = 0.0;
                continue;
            }
            if ( vh[j+1][k] > vh[j][k] ) {
                t = ar;
            }
            else {
                t = br;
            }
            if ( vf[j][k] < vf[j][k-1] ) {
                if ( vh[j][k-1] > vh[j+1][k-1] )
                    r = vh[j][k-1];
                else
                    r = vh[j+1][k-1];
                s = vf[j][k-1];
            }
            else {
                if ( vh[j][k] > vh[j+1][k] )
                    r = vh[j][k];
                else
                    r = vh[j+1][k];
                s = vf[j][k];
            }
            vy[j][k] = sqrt( vg[j][k]*vg[j][k] + r*r )* t/s;
            if ( (k+1) >= nz ) {
                vs[j][k] = 0.0;
                continue;
            }
            if ( vf[j][k] < vf[j-1][k] ) {
                if ( vg[j-1][k] > vg[j-1][k+1] )
                    r = vg[j-1][k];
                else
                    r = vg[j-1][k+1];
                s = vf[j-1][k];
                t = br;
            }
            else {
                if ( vg[j][k] > vg[j][k+1] )
                    r = vg[j][k];
                else
                    r = vg[j][k+1];
                s = vf[j][k];
                t = ar;
            }
            vs[j][k] = sqrt( vh[j][k]*vh[j][k] + r*r )* t / s;
        }
    }
```

This kernel looks complicated but there are no dependencies in it so it can be fully parallelized.

7.1.16 Kernels 16 and 17

The use of "`goto` inside those loops is causing unexpected behavior in our compiler. "Gotos" are not supported.

7.1.17 Kernel 18 - 2-D explicit hydrodynamics fragment

Large code, not listed here. However, all loops are fully parallel and parallelism is extracted as expected.

Table 2: Summarizing briefly the evaluation results for kernels 13–24

Kernel	*Expected Parallelism*	*Parallelism Extracted by C2μTC/SL*
13	serial execution	serial execution
14	1st: full parallelism	full parallelism
	2nd: serial execution	serial execution
	3rd: serial execution	serial execution
15	loop k: full parallelism	full parallelism
16		`goto` not supported
17		`goto` not supported
18	serial execution	serial execution
19	full parallelism	full parallelism
20	serial execution	serial execution
21	full parallelism	full parallelism
22	full parallelism	full parallelism
23	some parallelism	limited parallelism
24	serial execution	synchronizing memory

7.1.18 Kernel 19 – general linear recurrence equations

```
for ( k=0 ; k<n ; k++ ) {
    b5[k] = sa[k] + stb5*sb[k];
    stb5 = b5[k] - stb5;
}
for ( i=1 ; i<=n ; i++ ) {
    k = n - i ;
    b5[k] = sa[k] + stb5*sb[k];
    stb5 = b5[k] - stb5;
}
```

The variable `stb5` is a shared variable in both loops and using the shared variable mechanism we can accelerate the sequential execution of those loops.

7.1.19 Kernel 20 – Discrete ordinates transport

```
for ( k=0 ; k<n ; k++ ) {
    di = y[k] - g[k] / ( xx[k] + dk );
    dn = 0.2;
    if ( di ) {
        dn = z[k]/di ;
        if ( t < dn ) dn = t;
        if ( s > dn ) dn = s;
    }
    x[k] = ( ( w[k] + v[k]*dn )* xx[k] + u[k] ) /
            ( vx[k] + v[k]*dn );
    if (k>0) xx[k] = ( x[k] - xx[k-1] )* dn + xx[k-1];
}
```

Our compiler is taking a safe approach on cross dependencies between arrays in a loop (x depends on xx and xx on x) so this loop is considered to be sequentially executed.

7.1.20 Kernel 21 – matrix * matrix product

```
for ( k=0 ; k<25 ; k++ )
    for ( i=0 ; i<25 ; i++ )
        for ( j=0 ; j<n ; j++ )
            px[j][i] += vy[k][i] * cx[j][k];
```

This is a fully parallel loop and our compiler correctly understands it as such and extracts full parallelism out of it.

7.1.21 Kernel 22 – Planckian distribution

```
expmax = 20.0;
u[n-1] = expmax*v[n-1];
    for ( k=0 ; k<n ; k++ ) {
        y[k] = u[k] / v[k];
        w[k] = x[k] / (  y[k] -1.0 );
    }
```

Again a fully parallel loop and our compiler works as intended.

7.1.22 Kernel 23 – 2-D implicit hydrodynamics fragment

```
    for ( j=1 ; j<6 ; j++ ) {
        for ( k=1 ; k<n ; k++ ) {
            qa = za[j+1][k]*zr[j][k] + za[j-1][k]*zb[j][k] +
                 za[j][k+1]*zu[j][k] + za[j][k-1]*zv[j][k] +
                 zz[j][k];
            za[j][k] += ( qa - za[j][k] );
        }
    }
```

An anti-dependence and a dependence are found in this. If we consider the anti dependence as a real dependence (by inverting its vector in the dependence vector) then we had a dependence vector of 2 dependencies (1,0) and (0,1) and our compiler is using its dependence mechanism to produce a correct result with some parallelism in it.

7.1.23 Kernel 24 – find location of first minimum in array

```
x[n/2] = -1;
    m = 0;
    for ( k=1 ; k<n ; k++ )
        if ( x[k] < x[m] ) m = k;
l=m;
```

Here the `l=m` is introduced at the end of the loop so that the compiler will know that the value of m is needed after the loop has ended, m is decided to be a shared variable. Apart from that it is a standard sequentially executed loop with one shared variable.

7.2 Experimental Analysis

Experimental results for some interesting problems are presented in this section: matrix multiplication, grey-scale image smoothing and a loop with dependence vector (0,1)(1,0). Machine cycles and speedups achieved are presented for all these problems. All experiments compare the serial version (compiled by gcc and run on a single microthreaded core with no concurrency) with the version automatically generated by $C2\mu TC/SL$, which runs on SVP. All experiments have been performed using the rbm256 (256 cores) profile and the version 3.1.154-r4008 of the compiler and the simulator, running on 1,2,4,8,16,32 and 64 cores.

7.2.1 Matrix Multiplication

Let us first examine the speedup achieved for a very common and widely used application for such purpose: the matrix multiplication. The code we used follows. The code in SL has already been discussed in 6.2

```
for (i=0;i<N;i++)
    for (j=0;j<N;j++)
    {
        sum=0;
        for (k=0;k<N;k++)
            sum+=a[i][k]*b[k][j];
        c[i][j]=sum;
    }
```

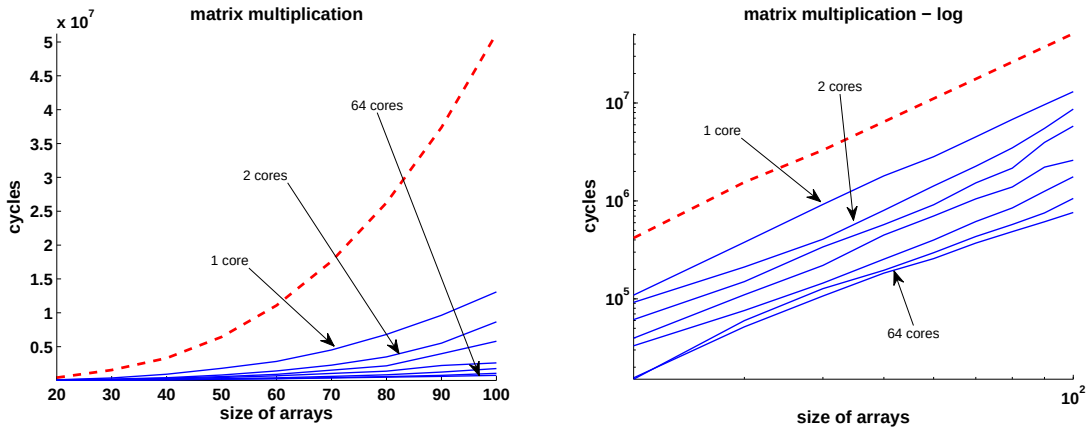


Figure 5: Matrix Multiplication. The dashed line is for the execution times for the serial version of the algorithm while the solid lines are for the automatically generated code running on SVP for 1,2,4,8,16,32 and 64 cores. The figure on the left presents the results in linear axis while the figure on the left is a log to log plot presenting the same information.

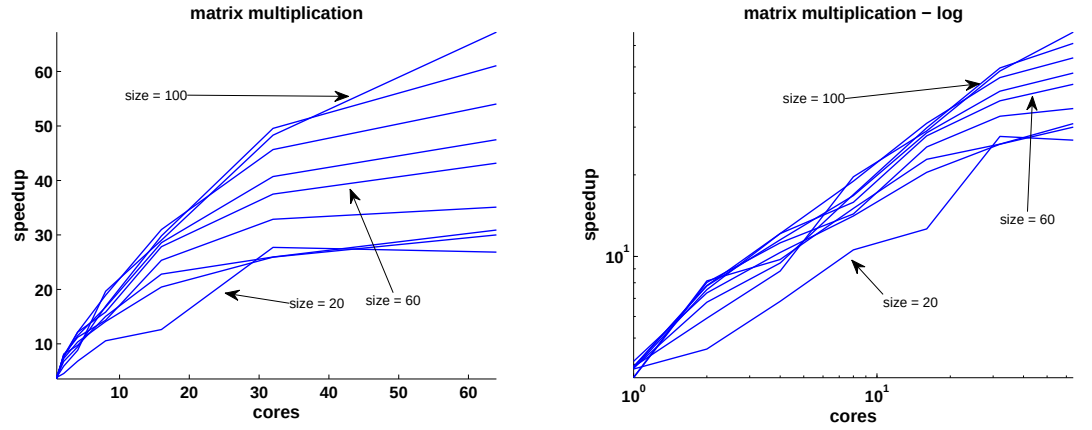


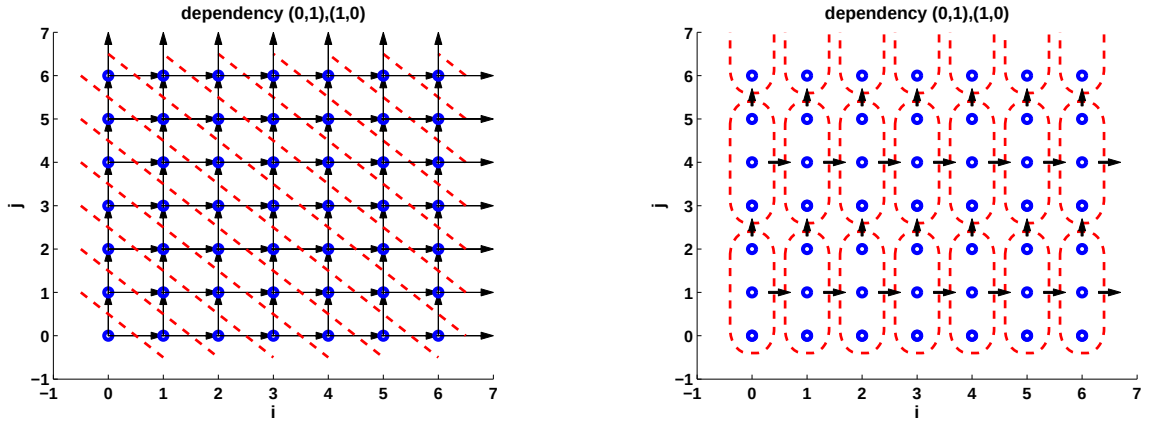
Figure 6: Matrix Multiplication. The dashed line is for the execution times for the serial version of the algorithm while the solid lines are for the automatically generated code running on SVP for 1,2,4,8,16,32 and 64 cores. The figure on the left presents the results in linear axis while the figure on the left is a log to log plot presenting the same information.

Figure 5 presents in linear and logarithmic axis the cycles necessary to compute the result when multiplying two arrays. The size of the arrays ranges from $N = 20$ to $N = 100$. In figure 6 the speedup is depicted again in both linear and logarithmic plots. The speedup achieved seems to be satisfactory and the execution times seems to scale well as the size of the problem increases.

7.2.2 Dependency (0,1),(1,0)

Now, let us now consider the following code:

```
for (i=1;i<N;i++)
    A[i][j]=A[i-1][j-1];
```



(a) The index space (b) Families of threads

Figure 7: A loop with two dependencies: (0,1),(1,0)

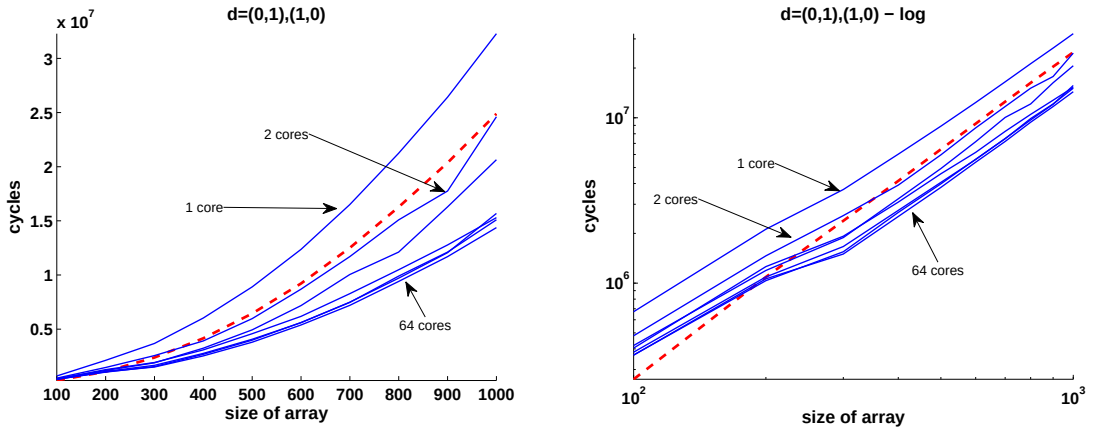


Figure 8: A loop with two dependencies: (0,1),(1,0). Dashed line is for execution times for the serial version of the algorithm while the solid lines are for the automatically generated code running on SVP for 1,2,4,8,16,32 and 64 cores. The figure on the left presents the results in linear axis while the figure on the left is a log to log plot presenting the same information.

This is a two single dependency loop in a two dimensional array. Each element $A_{i,j}$ of the array depends on the elements $A_{i-1,j}$ and $A_{i,j-1}$ as shown in figure 7(a). Families of threads are created and scheduled according to the hyperplane model as discussed in the previous sections and shown in figure 7(b).

The experimental results presenting machine cycles for different values of the size of the array ranging from $N = 100$ to $N = 1000$ can be found in figure 8 and the speedup in 9, in both linear and logarithmic plots. The overhead introduced by the run-rime scheduler for the initialization phase does not allow faster execution for small values of N and for execution on one core. However, for larger values of N and when using more than one core the execution on SVP is always faster and the trend implies even faster execution for larger values of N.

7.2.3 Image Smoothing

Finally, let us consider another very common source code. It finds application in grey-scale image smoothing, in the approximation of solutions of differential equations e.t.c. The code follows:

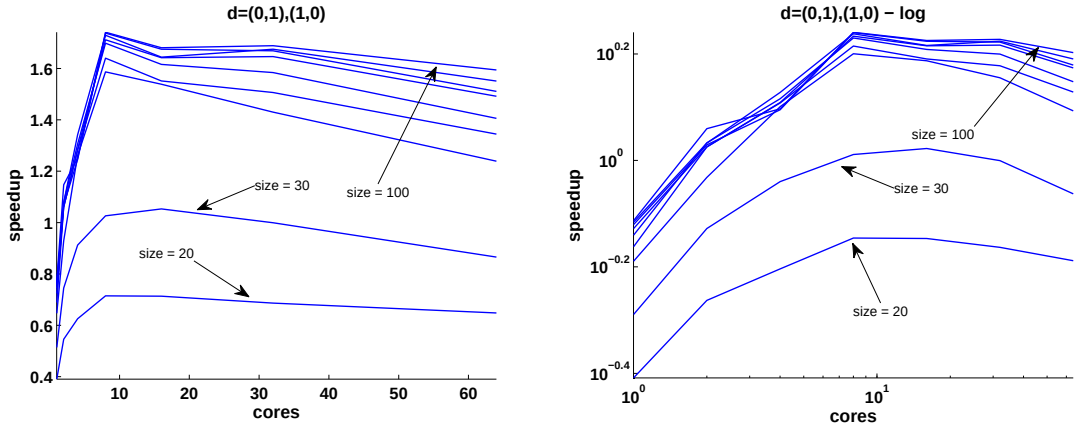


Figure 9: A loop with two dependencies: (0,1),(1,0). Dashed line is for execution times for the serial version of the algorithm while the solid lines are for the automatically generated code running on SVP for 1,2,4,8,16,32 and 64 cores. The figure on the left presents the results in linear axis while the figure on the left is a log to log plot presenting the same information.

```
for (i=1;i<n-1;i++)
    for (j=1;j<n-1;j++)
    {
        b[i][j]=a[i-1][j-1]+a[i][j-1]+a[i+1][j-1]+a[i+1][j]
            +a[i+1][j+1]+a[i][j+1]+a[i-1][j+1]+a[i-1][j];
        b[i][j]=b[i][j]/8.0;
    }
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        a[i][j]=b[i][j];
```

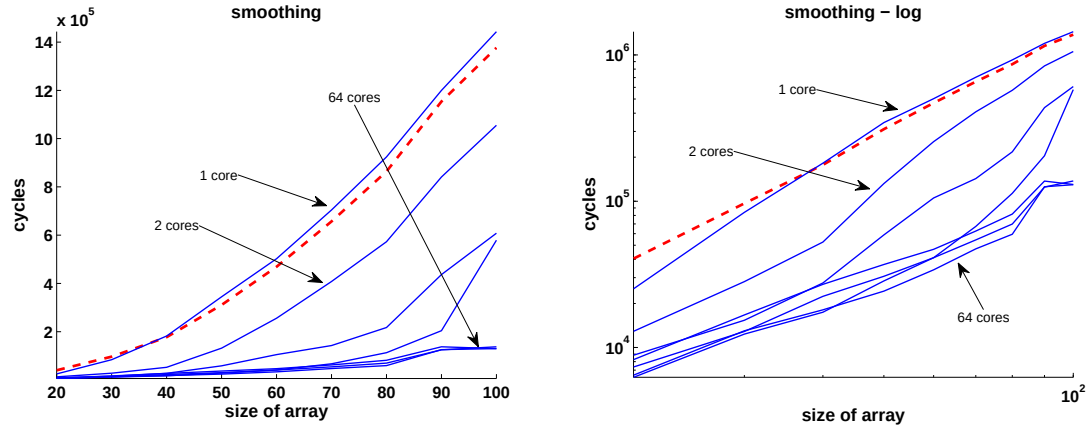


Figure 10: Grey-scale image smoothing. Dashed line is for execution times for the serial version of the algorithm while the solid line is for the automatically generated code running on SVP. The figure on the left presents the results in linear axis while the figure on the left is a log to log plot presenting the same information.

The experimental results are depicted in figure 10 and the speedup in figure 11, again in both linear and logarithmic plots. The speedup achieved is again satisfactory and executions on SVP are always faster than the serial one when more than one core is used.

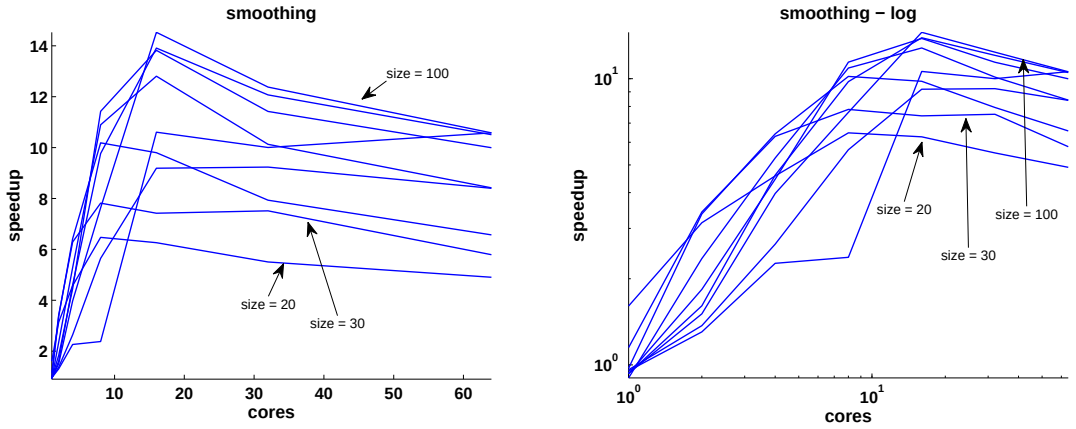


Figure 11: Grey-scale image smoothing. Dashed line is for execution times for the serial version of the algorithm while the solid line is for the automatically generated code running on SVP. The figure on the left presents the results in linear axis while the figure on the left is a log to log plot presenting the same information.

8 Conclusion and Future Work

$C2\mu TC/SL$ is a source to source C parallelizing compiler developed to exploit the special characteristics of the SVP processor. In this document the design and implementation of $C2\mu TC/SL$ is presented as well as some evaluation results including both qualitative and quantitative evaluation and showing that the automatic mapping of sequential programs on SVP is possible and can be done in an efficient way.

The next step is the evaluation of the compiler. Some evaluation results have already presented in this deliverable. We will also use the time until the end of the project to do some fine-tuning in the implementation of the compiler to succeed even better performance in the terms of the evaluation procedure. Some debugging is also necessary.

References

[1] ACE – Associated Computer Experts. CoSy compiler development system. Amsterdam, The Netherlands. http://www.ace.nl/cosy.html.

[2] K. Bousias, L. Guang, C.R. Jesshope, and M. Lankamp. Implementation and evaluation of a microthread architecture. *Journal of Systems Architecture*, 55(3):149–161, 2009.

[3] T. Bernard, K. Bousias, L. Guang, C.R. Jesshope, M. Lankamp, M.W. van Tol, and L. Zhang. A general model of concurrency and its implementation as many–core dynamic RISC processors. In *Proc. of International Symposium on Systems, Architectures, MOdeling and Simulation*, Samos, Greece, July 2008.

[4] C.R. Jesshope. *μTC*– an intermediate language for programming chip multiprocessors. In *Proc. of Pacific Computer Systems Architecture Conference 2006*, Shanghai, China, September 2006.

[5] Leslie Lamport. The parallel execution of DO loops. *Commun. ACM*, 17(2):83–93, 1974.

[6] F. H. McMahon. Livermore FORTRAN kernels: A computer test of numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, California, USA, December 1996.

[7] Xingfu Wu. *Performance Evaluation, Prediction and Visualization of Parallel Systems*, page 144. Springer, 1999.