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

C2μTC/SL - C Paralellizing Compiler targeting SVP Final Report

Deliverable D3.4, Issue 1.1

Workpackage WP3

Author(s):	George Manis		
Reviewer(s):	Chris Jesshope		
WP/Task No.:	WP3	**Number of pages:**	45
Issue date:	2011/07/29	**Dissemination level:**	Public

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

Approved by the project coordinator: Yes **Date of delivery to the EC:** 2011/07/29

Document history

When	Who	Comments
2011/06/16	George Manis	v1.0 – Initial version
2011/06/16	Chris Jesshope	Comments on v1.0
2011/07/28	George Manis	v1.1 – Final version

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

Table of Contents

1 Introduction

This document presents the 1st release of $C2\mu TC/SL$ [1,2] C parallelizing source to source compiler targeting the SVP [3–5] model. More specifically, it presents the transformations supported in this first release of the compiler, its novel run-time scheduler, the task parallelism supported for recursive functions, implementation details and an evaluation section presenting both qualitative and quantitative results. The source code of the 1st release of $C2\mu TC/SL$ is publicly available.

The implementation has been done using the CoSy[1] [6] compiler development system. The source language is the C programming language and the target language is SL, a script language extending C in order to express concurrency in the SVP model.

This is the final deliverable of the 3rd workpackage 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.

This document is written in order to be self-contained, updating previous reports and including the necessary background information.

2 $C2\mu TC/SL$ in the AppleCore Project

The purpose of the project is to develop the necessary infrastructure, compilers, operating system, execution platforms and tools in order to support and evaluate a novel architecture paradigm which can exploit many-core processors, the SVP model.

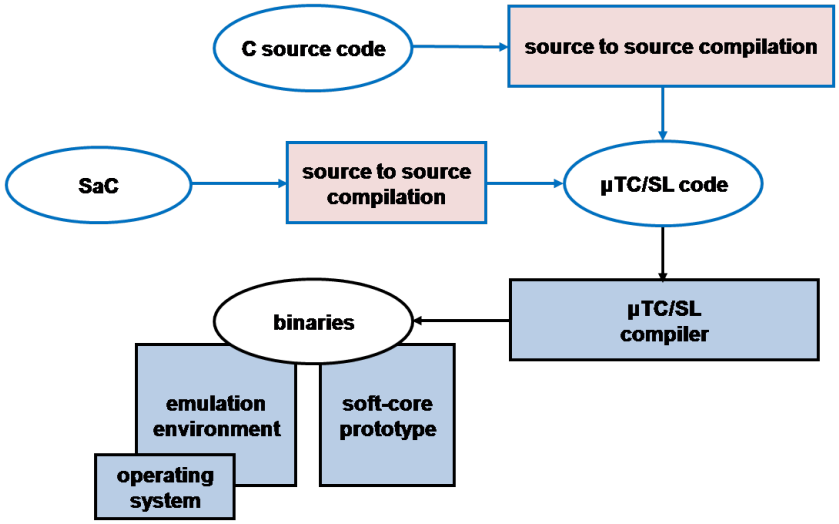


Figure 1: The AppleCore project

The structure of the whole project is shown in figure 1. Two parallelizing compilers have been developed. The $C2\mu TC/SL$ is a C parallelizing compiler which is mainly described in this deliverable and is the main deliverable for the 3rd workpackage. The target is to exploit the large amount of software already available in C. $C2\mu TC/SL$ tries to cover the procedural programming languages by selecting one of the most difficult ones to extract parallelism from. The SaC compiler is another parallelizing compiler for the SaC (Single Assignment C) [7] programming language. SaC targets the functional programming parallelism. Both $C2\mu TC/SL$ and SaC compilers are source to source compilers which produce code in μTC [8] or SL [9] languages. μTC and SL are extensions of C which exploit the special characteristics of SVP. Programs written in μTC and SL are compiled to binaries by the $\mu TC/SL$ compiler. These binaries can be executed in an emulation environment. An FPGA prototype is also available.

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

Three academic partners were involved. The University of Amsterdam was the leader partner, coordinated the project and was mainly responsible for the design of the SVP model and the languages μTC and SL. In the University of Amsterdam the emulation environment, the $\mu TC/SL$ compiler and the operating system were also developed. The SaC compiler was constructed in the University of Hertfordshire and the $C2\mu TC/SL$ compiler in the University of Ioannina, Greece. The softcore prototype was developed in UTIA, in Prague.

Two industrial partners were also involved. ACE is a Dutch company that provided CoSy, a compiler development framework. The $C2\mu TC/SL$ compiler has been developed on top of CoSy. The flexibility, reliability and high quality of the CoSy framework and the experience and knowhow of the company was proved an invaluable aid to the development of $C2\mu TC/SL$. The second industrial partner was Aeroflex Gaisler, a Swedish company involved in the development of the softcore prototype.

3 SVP: The Self-Adaptive Virtual Processor and Model

The SVP model is a general concurrency model not only targeting special purpose but also generic applications. In this brief description we will describe and consider this model as an abstract execution model.

The unit of parallelism is a *thread*. Threads are organized in groups which are called *families*. Each thread can create new families, i.e. families are created by threads belonging in other families. This makes the model hierarchical. The hierarchical structure of SVP is shown in figure 2.

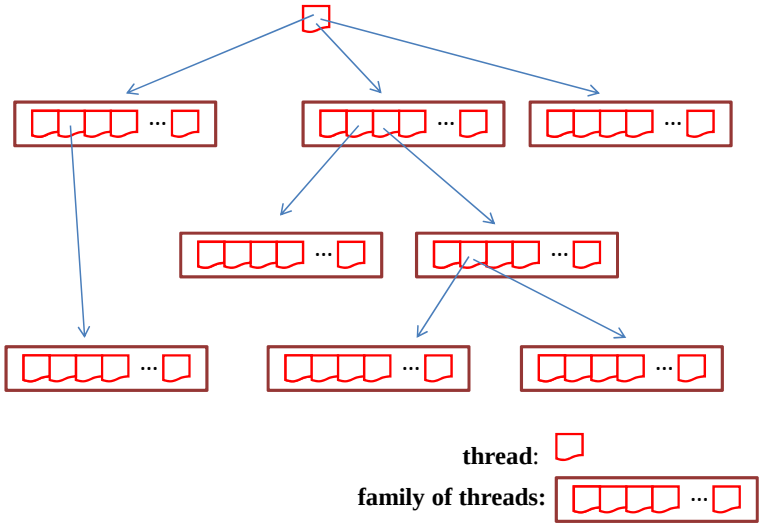


Figure 2: The hierarchical structure: threads are grouped in families and families are created by threads

Threads in a family are ordered, i.e. each thread has a unique number which identifies the order of the thread in the family. This unique number not only identifies the thread but also defines the communication channels between threads belonging in the same family. More specifically, the communication between threads can be done in two different ways: (i) using the *shared memory* and using (ii) the *synchronizing memory*.

All threads have access to the shared memory. The consistency model is the simplest possible, i.e. there is no guarantee for any race conditions when two or more threads attempt to access the same memory location simultaneously. Memory is made consistent at create and sync points for a family and communication is only guaranteed between parent and child threads and child threads and their parent.

The synchronizing memory allows unidirectional communication between thread with index i and thread with index i+1. This memory simulates unidirectional synchronization channels between

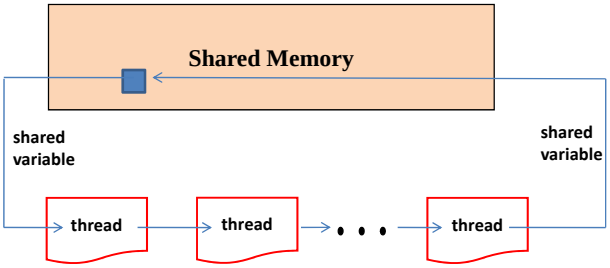


Figure 3: Unidirectional communication in the SVP model

successive threads, carrying not only synchronization signals but also data (fig. 3). Synchronizing memory can also be used to speed up the access to memory locations which are read-only for all threads in the family.

No other kind of communication is possible between any two executing threads. There is no way for threads belonging in different families to communicate with each other using the synchronizing memory, at least in the user programs level. Given the above restrictions for the consistency of the shared memory, the communication between threads belonging to different families is at least not efficient and is usually better to be avoided. Figure 4 presents schematically the processing model.

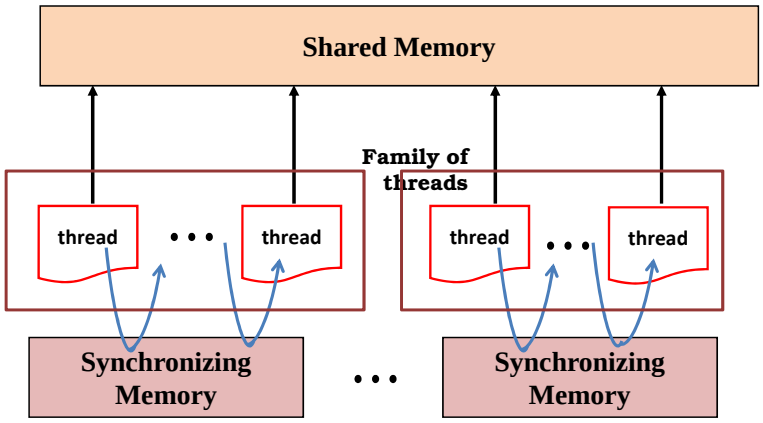


Figure 4: The processing model: threads, families, memories and unidirectional communication

We will give an example to show how we can exploit parallelism in SVP. Let us consider the inner product. What SVP achieves here is to capture sequence through dataflow constraints between concurrent threads.

```
sum=0;
for (i=0;i<N;i++)
    sum+=a[i]*b[i];
```

We need one family consisted of N threads of execution. Each thread i will perform the multiplication `a[i]*b[i]` and then wait to receive the partial sum for the thread i-1, namely it will receive the amount $sum = \sum_{j=0}^{i-1} a[j]b[j]$. After receiving the partial sum from the previous thread, it will add its own contribution and forward the result to thread $i+1$, namely it will forward the amount $sum = \sum_{j=0}^{i} a[j]b[j]$. All loads and multiplications `a[i]*b[i]` will be scheduled independently on one core and will be performed in parallel and only the summation of the multiplications will be serial. For the summation procedure the synchronizing memory will be used which allows fast forward of data from thread i to thread $i+1$. The execution times for the serial execution and the execution on SVP are shown in figure 5.

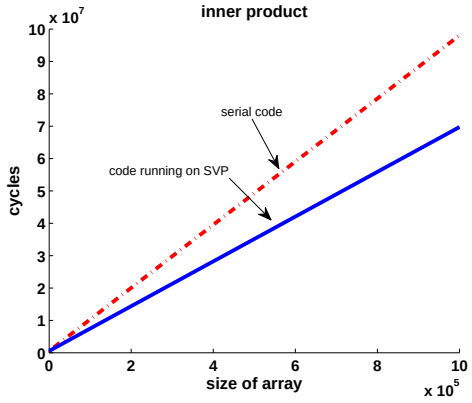


Figure 5: Execution times for the inner product in one core.

4 Expressing Concurrency in Language Level: μTC and SL

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 thread 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 which masks more details from the programmer and provides a more friendly and familiar user interface to the programmer. Let us see briefly the main constructs of the language.

`sl_create(,,from,to,step,,,thread, ...)` − Creates a new family of $\lfloor \frac{to-from}{step} \rfloor$ threads. Each thread executes the function `thread`. The value `from` is assigned to the index of the first thread, the value `from+step` is assigned to the second thread and so on.

`sl_index(variableName)` − Declares which variable is the index of the family.

`sl_sync()` − A barrier which waits until all threads in a family complete execution.

`sl_def(thread, void, ...){codeInC} sl_enddef` − Defines the thread `thread` which can be used by `sl_create`. The code of the function is written in C language and is included between `sl_def` and `sl_enddef`.

`sl_shparm(parameterType,parameterName)` − Definition of a variable as *shared*.

`sl_glparm(parameterType,parameterName)` − Definition of a variable as *global*. A *global* variable is a variable which is read-only for all the threads in the family.

`sl_sharg(argumentType,argumentName,argumentValue)` − Passing value (initialization) to a variable defined earlier as shared with the `sl_shparm`.

`sl_glarg(argumentType,argumentName,argumentValue)` − Passing value (initialization) to a variable defined earlier as global with the `sl_glparm`.

`sl_setp(sharedParameterName,localVariable)` − Copies the value of a local variable to a shared variable. It can be called only once for each shared variable.

`localVariable=sl_getp(sharedParameterName/globalParameterName)` − Copies the value of a shared/global variable to a local variable. It can be called only once for each shared/global variable.

`variable=sl_geta(sharedArgumentName)` − Used when a family completes execution, to get the value of a shared variable.

`sl_break()`− Terminates the family in which the calling thread belongs.

The following code is written in SL and computes the factorial:

```
sl_def(thread, void, sl_shparm(int,  _s))
{
  int s;
  sl_index(x);

  s=sl_getp(_s);    -- gets s from previous thread
  s=x*s;            -- updates s
  sl_setp(_s,s);    -- forwards s to the next thread
 }
sl_enddef

sl_def(t_main,void)
{
   int N=15;
   sl_create(,,1,N+1,1,,,thread,sl_sharg(int, _s, 1));  -- creates a family
   sl_sync();  -- waits until all threads in the family terminate
   printf("%d",sl_geta(_s)); -- gets the value of s and prints it
}
sl_enddef
```

5 The $C2\mu TC/SL$ source to source compiler, a quick overview

$C2\mu TC/SL$ is a parallelizing compiler which maps sequential C programs on the SVP concurrent processing model. The aim of $C2\mu TC/SL$ is to extract parallelism from C programs and produce families of threads in order to exploit the special characteristics of SVP. Since we expect to extract most of the parallelism from loop structures, $C2\mu TC/SL$ mainly focuses on loops. It can also extract parallelism from some recursive functions. Contrary to most parallelizing compilers, $C2\mu TC/SL$ moves the problem of scheduling from compile-time to run-time: at compile-time a lightweight scheduler is generated which in runtime will be responsible for the coordination of the execution of the families allowing more flexibility than static scheduling. Let us have a short description of how $C2\mu TC/SL$ handles loops. The compiler detects parallelism inherent in loop structures (for and while loops) and maps them on the underlying processing model as shown on figure 6.

A number of threads dependent on the limits and the step of the loop are organized as a family. Each thread executes the same code but has a different index, something that can differentiate the execution on each thread. The compiler detects which data has to move from thread to thread as shared variables, which data are local in a thread, and which data will remain in the main memory. Shared variables are initialized in the first thread. As an example, consider the following series, computing the amount:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^n}{n!}$$

Let us assume that for the computation of the above series we write the following code:

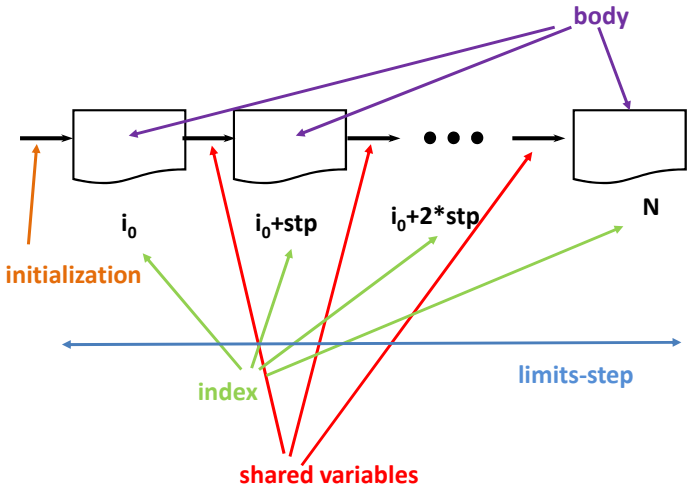


Figure 6: A loop structure mapped on SVP model.

```
e=1;
for (i=1;i<=x;i++)
{
   p=pow(x,i);
   f=fact(i);
   e+=p/f;
}
```

The compiler will produce one family of x threads. Each thread will be responsible for one of the terms of the summation. All computation will take place in parallel until the summation is reached. A shared variable initialized to 1 will pass from all threads in the family starting from the first, collecting and accumulating the terms to the total sum. Since the part of the computation before the dependency can be performed in parallel and since the summation is done using the synchronizing memory and shared variables, the version running on SVP is much faster than the serial one.

6 The Tool Chain

The compilation procedure for producing binaries from the C input file is shown in figure 7. 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).

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):

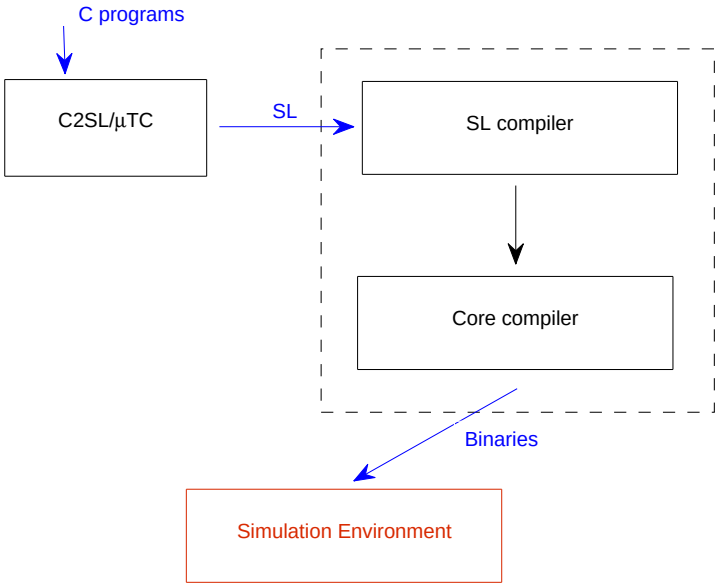


Figure 7: From a C program to the program output

```
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
```

where `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
```

7 Mapping Loops on SVP with $C2\mu TC/SL$

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.

7.1 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 at the dependency. To minimize the delay introduced by the dependency, the synchronizing memory is used and a shared variable is created.

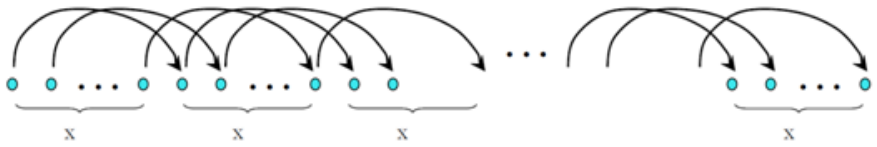


Figure 8: A simple loop, not necessarily unary, with one dependency.

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 (fig. 8). 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 $(i-1)$ (fig. 9).

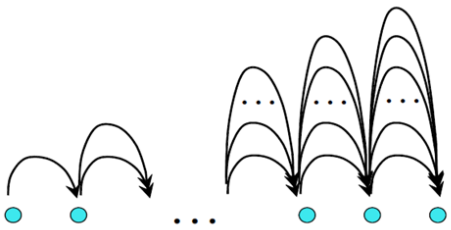


Figure 9: A simple loop with more than one dependencies.

7.2 Nested Loops

The transformations of nested loops are not just a straightforward generalization of those of simple loops.

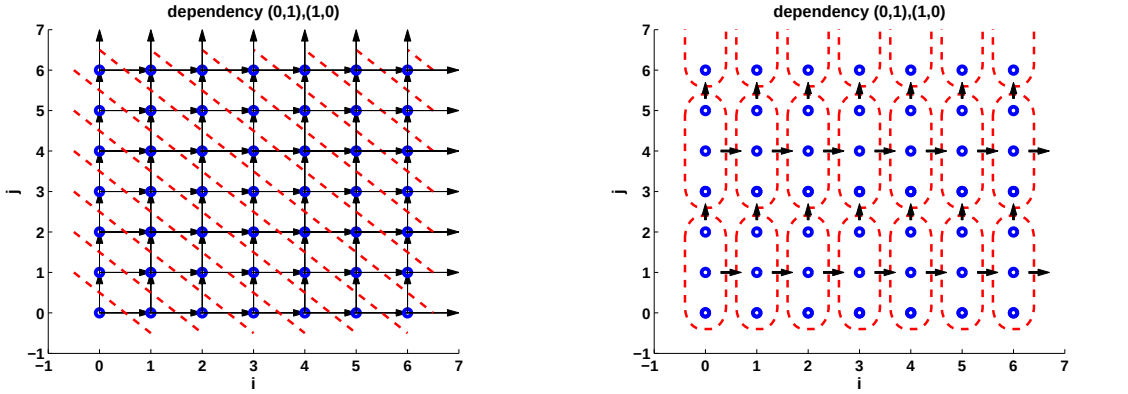
In the following example we need to handle two dependencies ($a > 0,\ b > 0,\ c > 0,\ d > 0$).

```
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 [10], 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 [10] we can find which points can be computed in parallel. In figure 10(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 10: From the hyperplane to the concurrent families of threads

The solution is 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 10(b). Dashed boxes are families of threads. Upon finishing execution, a family of threads 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.

7.3 "While" Loops

`While` loops are much more difficult to be handled than `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.

8 Scheduling at Run-time

The option to schedule tasks at run-time has not been studied a lot, at least as an option of the compiler to produce code for managing the execution of the parallel application. However, run-time scheduling has many advantages than cannot be ignored. Advantages of run-time scheduling include the flexibility to detect non-static dependencies, to use index spaces with no predetermined or irregular limits and to avoid mathematically complicated transformations. The main drawback of scheduling tasks while the application is running is that the scheduler shares resources and computational time with the rest of the tasks performing computations. In an application in which the available resources are underutilized this is not a problem, since the scheduler can run without delaying the rest of the application. This is a very common situation for many applications which are not fully parallel and present a kind of limited fine grain concurrency. Usually the execution of this kind of applications has much to benefit when running on multicore systems, compared to running on traditional architectures.

Existing compile-time techniques rely on a set of heuristic methods to solve a linear programming problem (calculating the transformation matrix of the coordinates) and even if they do get a good solution to it, still need to find a reverse transformation system between the original coordinates of the loop indexes and the transformed ones. Our system tackles this problem by discovering the emerging hyperplanes as execution goes on without the need for expensive transformations. Working at run-time also gives us the ability to work with loops with irregular index spaces. In the experimental results section we perform experiments showing that the only disadvantage of run-time scheduling, the overhead introduced by the scheduler, is reasonable and does not pose an obstacle for its use.

The general idea of our algorithm is simple. If at any given moment we know which indexes are being computed, then we can apply the dependence vector on the index vector and come up with a new index vector where if the total number of dependencies converges on an element of it, that element needs to be computed in the next step. For that to work efficiently we are tiling the elements of the innermost loop dimension in groups (families). Parallelization takes place between families and code execution inside each family happens sequentially.

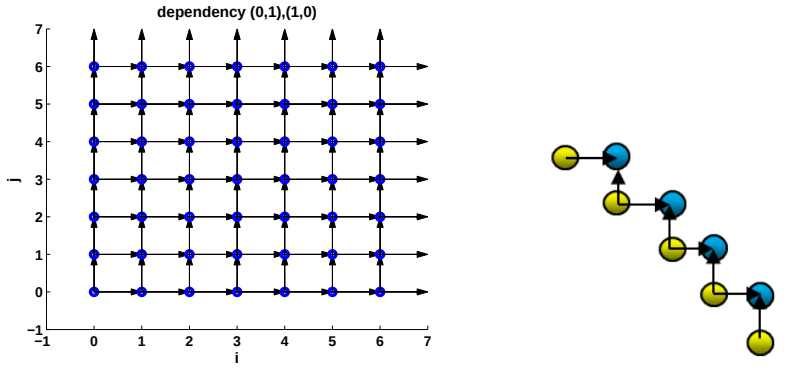
Please consider the following example code:

```
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 11(a).

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 11(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



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

(b) The movement of data

Figure 11: Run-time scheduling example

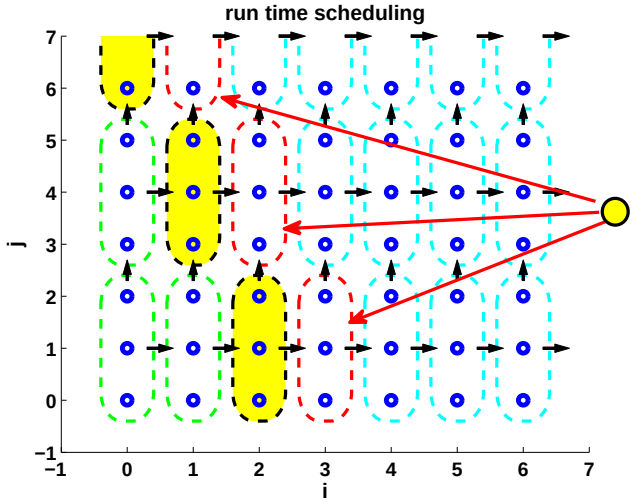


Figure 12: The run-time scheduler.

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 12 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.

Our algorithm is a two-step process. First is the initialization and second is the actual execution of the code. After the number of families has been decided, the initialization part creates an array of size equal to the total number of families in our search space and initializes it to 0. For that we use the SVP system to achieve parallelism during that part. The array is a single dimension one so it is rather straightforward to set all its elements to 0 with a single create call.

After that we need to decide which families will begin execution during the execution part of the algorithm. This info will come by knowing the dependence vector. Vector is analyzed per dimension and different code is produced depending on the size of the dependency on that particular dimension. If the dependency on a particular dimension is 0 then this means that this dimension will be in parallel completely otherwise parallelism extends from 0 to the size of the dependency minus 1. Extra care needs to be given to negative numbers. What we end up in the end is a piece of code

in the shape of a nested loop. The outer loops are all the dimensions where the dependencies are 0 and inside the rest of the loops. In the loop body we set the array we initialized before to max number of dependencies satisfied and add the coordinates of the indexes into our vector of families to begin computation.

Effectively the array we created keeps track of how many dependencies per computation element (family) are satisfied at any given cycle. Once the computation begins, a family of threads is created, its size equal to the total number of elements to be computed plus one (n). While the n threads are executing the computation, the last thread is going through the vector of index coordinates and adds on each element of it, the elements of the dependence vector. For each new coordinate element, its corresponding position in the array we created is increased by one. If that number reaches the total number of dependencies in our dependence vector, then that element is added on another vector which contains the elements that will be executed on the next cycle (called the next vector).

Once all the threads finish their execution, the next vector becomes the computation vector and the cycle continues. This goes on until the next vector returns empty. At that point, it is considered that there is nothing left to compute and the computation stops. Schematically in the form of pseudocode the whole procedure looks like this:

```
- divide the last dimension of the loop by the tiling factor
    (number of families)
- allocate memory for the satisfied dependency array (equal to the
    product of the sizes of all dimensions save the innermost one
    where the divided number is used)
- initialize that array in parallel
    - create a family of threads equal to the size of the array
    - index i
    - array[i]=0
- code that adds the families of threads that will begin execution
    in the computation vector
    - it's dependent on the dependency vector.
    - if a dependency on a given dimension is 0 then the whole
        dimension is executed in parallel
    - based on the value of the dependency on the dimension all
        indexes from 0 to that value -1 can be executed in parallel
- an endless loop starts here
- assuming that the computation vector's size is n:
- create a family of n+1 threads
    - index i
    - if  i<=n
        - create a family of threads equal to the size of the
            innermost dimension divided by the number of families
        - execute code sequentially
        - serialization is achieved by using one shared variable
    else
        -  loop through the indexes of the computation vector
            - for each set of index coordinates
                - loop through the dependence vector
                    - add the current dependency on the index
                        coordinates under consideration
                    - if the new set of coordinates is inside the
                        index grid bounds, then increase that place
                        on the array by one
                    - compensate for dependencies that are not
                        accounted for due to the coordinates being
                        near the boundaries of the grid space
                    - if the value of the array on that place equals
                        the number of total dependencies add the new
                        index coordinates to the next vector
```

```
    synchronize all threads
if the next vector is empty then stop computation
    else
    set the pointer of the computation vector to the next vector
    set the next vector to empty
```

9 Implementation

The *implementation* section has also been included in D3.3 and is added here without any modifications, for completeness.

9.1 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 7.3.

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` structure 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.

9.2 Illustrative Examples

9.2.1 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
```

9.2.2 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 masterloop i:

```
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);
```

10 From Cetus to SL

In parallel with the development of $C2\mu TC/SL$ compiler during the last period of the project we developed another compiler which exploits the ability of Cetus [11–13] to extract parallelism. We selected Cetus since it is one of the most active parallelizing compilers today and the source code was available on the Internet.

We downloaded, installed and deeply investigated Cetus, both the available documentation as well as the source code in order to decide which of the facilities of Cetus were useful for mapping C onto SVP. We found several interesting features: among other, Cetus can detect full parallel loops, reductions, and can characterize loop variables according to the way they accessed.

We developed code that reads the output of Cetus exploiting all the above facilities and maps it on SVP. This code has been developed on Lex and Yacc and not on top of CoSy like $C2\mu TC/SL$. We will call this compiler $Cetus2\mu TC/SL$ [14].

In order to avoid long descriptions we will present $Cetus2\mu TC/SL$ using examples. The first example shows how Cetus detects a full parallel loop and how $Cetus2\mu TC/SL$ maps it on SVP. The code in C follows:

```
int main()
{
   int i, n;
int a[10], b[10], c[10];
   n = 10;
   for (i = 0; i < n; i++)
      a[i] = b[i] + c[i];
   return (0);
}
```

After analysis with Cetus the code becomes:

```
int main()
{
   int i, n;
   int a[10], b[10], c[10];
   int normalizeReturn_val_0;
   n=10;
   #pragma cetus private(i)
   #pragma loop name main#0
   #pragma cetus parallel
   for (i=0; i<n; i ++ )
      a[i]=(b[i]+c[i]);
   normalizeReturn_val_0=0;
   return normalizeReturn_val_0;
}
```

Cetus recognizes that the loop can be executed in parallel. It also understands that `i` is a private variable in each iteration. $Cetus2\mu TC/SL$ reads this information from Cetus output and produces the following code, which is actually a fully parallel `create`:

```
typedef int *t1_0;
sl_def(thread0, void, sl_glparm(t1_0, _a),
   sl_glparm(t1_0, _b), sl_glparm(t1_0, _c))
{
   t1_0 a;
   t1_0 b;
   t1_0 c;
   sl_index(i);
   a = sl_getp(_a);
   b = sl_getp(_b);
   c = sl_getp(_c);
   a[i] = (b[i] + c[i]);
}
sl_enddef
```

```
sl_def(t_main, void)
{
   int i, n;
   int a[10], b[10], c[10];
   n = 10;

   {
      sl_create(, , 0, n, 1, , , thread0, sl_glarg(t1_0, _a, a),
         sl_glarg(t1_0, _b, b), sl_glarg(t1_0, _c, c));
      sl_sync();
   }
   return (0);
}
sl_enddef
```

The second example shows a reduction. Cetus detects the reduction and $Cetus2\mu TC/SL$ creates a family of threads to execute the operation. The following C code

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i] * b[i];
```

when analyzed by Cetus produces:

```
#pragma cetus private(i)
#pragma loop name main#0
#pragma cetus reduction(+: sum)
#pragma cetus parallel
#pragma omp parallel for private(i) reduction(+: sum)
for (i=0; i<n; i ++ )
sum+=(a[i]*b[i]);
```

Cetus recognizes that `i` is a private variable for each iteration, recognizes that there is a reduction on `sum` and that there is some parallelism on this loop. $Cetus2\mu TC/SL$ reads this information and produces the following code which assigns the loop to a family and the reduction to a shared variable:

```
typedef int *t1_0;
sl_def(thread0, void, sl_glparm(t1_0, _a), sl_glparm(t1_0, _b), sl_shparm(int, _s0))
{
   t1_0 a;
   t1_0 b;
   int sum;
   sl_index(i);
   a = sl_getp(_a);
   b = sl_getp(_b);
   sum = sl_getp(_s0);
   sum += (a[i] * b[i]);
   sl_setp(_s0, sum);
 }
sl_enddef

sl_def(t_main, void)
{
   int i, n, sum;
   int a[10], b[10];
   int normalizeReturn_val_0;
   n = 10;
   sum = 0;
```

```
    {
       sl_create(, , 0, n, 1, , , thread0,
               sl_glarg(t1_0, _a, a), sl_glarg(t1_0, _b, b),
               sl_sharg(int, _s0, sum));
       sl_sync();
       sum = sl_geta(_s0);
    }
  }
  sl_enddef
```

Finally Cetus identifies variables which are private in each iteration but their values are used after the loop as well. An example follows:

```
int main()
{
   int i, x, value;
   value = 10;
   for (i = 0; i < 10; i++)
   {
      x = (i - 1) * (i - 2) * (i - 3) * (i - 4) * (i - 5);
      value = 2 * x + value;
   }
   value = x * (value - 1);
   return (0);
}
```

The variable `x` is private in each iteration, but its value is used after the end of the loop as well. Cetus characterizes this variable as *lastprivate*:

```
int main()
{
int i, x, value;
int normalizeReturn_val_0;
value=10;
#pragma cetus lastprivate(x) private(i)
#pragma loop name main#0
#pragma cetus reduction(+: value)
#pragma cetus parallel
for (i=0; i<10; i ++ )
{
   x=(((((i-1)*(i-2))*(i-3))*(i-4))*(i-5));
   value=((2*x)+value);
}
value=(x*(value-1));
normalizeReturn_val_0=0;
return normalizeReturn_val_0;
}
```

$Cetus2\mu TC/SL$ uses a shared variable for `x`:

```
sl_def(thread0, void, sl_shparm(int, _s1), sl_shparm(int, _s0))
{
   int x;
   int value;
   sl_index(i);
   x = (((((i - 1) * (i - 2)) * (i - 3)) * (i - 4)) * (i - 5));
   value = sl_getp(_s0);
   value = ((2 * x) + value);
   sl_setp(_s0, value);
```

```
   sl_getp(_s1);
   sl_setp(_s1, x);
}
sl_enddef

sl_def(t_main, void)
{
   int i, x, value;
   value = 10;
   {
   sl_create(, , 0, 10, 1, , , thread0,
      sl_sharg(int, _s1, x), sl_sharg(int, _s0, value));
   sl_sync();
   x = sl_geta(_s1);
   value = sl_geta(_s0);
   }
value = (x * (value - 1));
return (0);
}
sl_enddef
```

11 Recursive Functions

It is common practice for the programmers to write recursive programs without taking into account the overhead introduced by the successive function calls or from possible repetition of computations. Compilers often use techniques like tail recursion in order to speed up the execution of recursive functions. Parallelization techniques have also been appeared in which the compiler tries to extract concurrency from the recursive function calls. Much work has been done in this field since recursive functions usually hide a high degree of parallelism.

In the very popular divide and conquer technique some input is divided into two or more parts and a recursive call is invoked for each one of those parts. Usually each call operates on a different part of the data and, thus, it is safe to perform these recursive calls in parallel. For example, let us consider the quicksort algorithm. The array is divided into two smaller arrays, and one recursive call is performed for each one of the smaller arrays. Since each call performs on different data it is safe to invoke both calls in parallel. This is an easy task for a parallel programs developer since it can be easily understood that each call performs on different data. The problem becomes difficult for a compiler, where the detection of parallelism must be automatic. A framework for automatic parallelization of such recursive functions is discussed in [15] where the main target is the divide and conquer algorithms. At compile, time symbolic array section analysis is used to detect the interdependence of multiple recursive calls in a procedure. When the compile time analysis is not enough, speculative run time parallelization is employed. Similar work is described in [16] where parallelism from recursive functions targeting divide and conquer algorithms is also extracted. In [17] the Huckleberry tool is presented, again detecting parallelism from divide and conquer algorithms and producing code for a multi-core platform. A quantifier-elimination based method belongs in the same family too. This method shrinks function closures representing partial computations, splits the input structure and perform computation on each part in parallel [18].

However, when a recursive call does not call more than one instance of itself, the extraction of parallelization is more difficult. Let us consider the factorial as an example. The computation of `fact(n)` requires that the computation of `fact(n-1)` has been completed. Thus, the kind of parallelism discussed above cannot be detected here. We target this kind of recursive function on which the commonly used methods cannot be applied and try to exploit threads of execution in order to achieve a finer lever of parallelism in the instruction level.

In similar work, we must also mention recursion splitting [19], a technique which converts recursive functions into loops and then applies loop parallelization techniques on them. In [20] an

analytical method is presented which transforms recursive functions on general recursive data structures into compositions of parallel skeletons. Both papers use functional languages.

Recursive function parallelism has been implemented as part of the $Cetus2\mu TC/SL$ parallelizing compiler and has been described in detail in [14] and will be presented in [21].

A recursive function call consists of successive calls of the same function each one of which performs on different data. These data have been produced by previous calls of the function in the chain of the function calls, or are common for all calls. In most cases each function uses the data produced by its calling functions, or at least uses data transferred through one of its calling functions.

When mapping a recursive function onto SVP, a family of threads is created and each one of the successive recursive calls is assigned to one thread of execution. Threads in a family are ordered. Each thread i in the family can communicate with the thread i+1 and forward data in a way similar to the "return" call of the recursive function. Obviously, the first thread in the family corresponds to the last function call in the chain of the recursive functions, the second thread to the one before the last, etc.

The information needed to be extracted is how many threads are to be created and what kind of communication is required between them. Of course, the great challenge is to do this automatically, without any hint from the programmers. In case we fail to extract this information it is better not to attempt to parallelize the execution at all and choose to run the program sequentially.

The structure of a recursive function call is not the same for all problems. However we can observe that in most cases this structure is of the form:

```
if (...) return(...)
else if (...) return(...)
else if (...) return(...)
...
else return(...)
```

When the structure of the function is as shown above, it is possible to decide how many threads have to be created and what kind of communication is necessary between them. The steps we have to follow are:

1. we decide which variable can be used as an index. In most recursive functions such a variable exists and can be located in the conditions of the `if` statements and in the parameters of the function

2. we detect for which values of this variable it is not necessary to call the function recursively

3. we detect how the value of this variable changes for any two successive calls in the chain of the recursive calls

4. based on steps 2 and 3, we identify how many recursive calls will be done. We create a family with the same number of threads

5. we decide on what data should move from one thread to the next thread in the family

6. we decide on what data are global to all threads, i.e. are not modified in the body of the function call

7. we detect the initial values for the variables of the first threads in the family

8. we compile the recursive function and produce the SL code

Let us consider a small example here, the computation of the factorial:

```
long int factorial(int x)
{
    if (x<0) return(-1); // not defined
    else if (x>L) return(-2); //overflow
    else if (x==0) return(1);
    else return(x*factorial(x-1));
}
```

According to the steps discussed above, the following information must be extracted.

1. the variable which can be considered as the *index* is x. The variable x exists in all conditions and in the parameters of the function
2. the function will not perform any recursive calls when $x < 0$ or $x > L$
3. each recursive call is based on data returned by the next recursive call. The value of x is reduced by 1 in every call, according to the $x-1$ actual parameter in the call
4. suppose we want to compute the factorial of N. Based on steps 2 and 3 we can conclude at compile time that we need N recursive calls. Thus, we must create a family with N threads
5. each thread forwards to the next thread in the family an integer, which is equal to the return value of the corresponding function call
6. the first thread is initialized to 1, the return value of the function for $x = 0$. This value is found if we move downwards (as indicated in step 3) from the given value until we found a value the computation of which does not require a recursive call
7. the information extracted in the steps above is enough to automatically produce a program in SL which produces the same results with the function `factorial` and can execute in parallel exploiting the special characteristics of the SVP architecture

The expected speedup of mapping such a recursive call onto the SVP computing model can be summarized in the following steps:

1. the execution environment for each thread is created in parallel and not sequentially as in recursive calls
2. the part of the computation which performs the evaluation of the condition is executed in parallel, since the value of x is given for each thread from the beginning (x is the index of the thread inside the family)
3. the rest of the computation, before the recursive call, can also run in parallel. In this example, x can move to the register and wait for the second operand before the multiplication can be performed. Please note that this part of the computation can be more expensive or much expensive. For example, if we had x^3 or $\sqrt{x}$ or generally $f(x)$ instead of x the computation before the recursive calls becomes more expensive.

Let us now discuss another one interesting example, the fibonacci numbers:

```
long int fib(int x)
{
    if (x<0) return(-1); // not defined
    else if (x>L) return(-2); //overflow
    else if (x==0) return(1);
    else if (x==1) return(1);
    else return(fib(x-1)+fib(x-2));
}
```

In this example the function `fib` is called recursively twice. We can again exploit the synchronizing memory and employ two shared variables this time, one for keeping the fibonacci number computed by the thread $i-1$ and one for the fibonacci number computed by the thread $i-2$. In this way we eliminate the need for each function call to call itself twice. This leads to a significant reduction of the necessary number of threads and to a further improve of the execution time, even more significantly this time as we will see in the following section.

12 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 6.

12.1 Parallelism Extraction from Livermore Loops

The qualitative evaluation based on Livermore Loops was also presented in D3.3. It has been added in this report without any modifications, for completeness.

The Livermore Loops is a benchmark for parallel computers. It consists of 24 kernels. The benchmark was published in 1986 [22, 23]. 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.

12.1.1 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.

12.1.2 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.

12.1.3 Kernel 3 – inner product

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

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 $C2\mu TC/SL$.

12.1.4 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}.

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

12.1.5 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.

12.1.6 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.

12.1.7 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.

12.1.8 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.

12.1.9 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.

12.1.10 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.

12.1.11 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.

12.1.12 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.

12.1.13 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.

12.1.14 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.

12.1.15 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;
```

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

```
            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.

12.1.16 Kernels 16 and 17

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

12.1.17 Kernel 18 - 2-D explicit hydrodynamics fragment

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

12.1.18 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.

12.1.19 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.

12.1.20 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.

12.1.21 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.

12.1.22 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.

12.1.23 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.

12.2 Experimental Analysis with Loop Based Benchmarks

Experimental results for some interesting problems are presented in this section: matrix multiplication and a nested loop with dependence vector (0,1)(1,0). Machine cycles and speedups achieved are presented. 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.

Part of this section was included in D3.3. However, the experimental results have been updated as necessary.

12.2.1 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 9.2

```
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 13 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 14 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.

12.2.2 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];
```

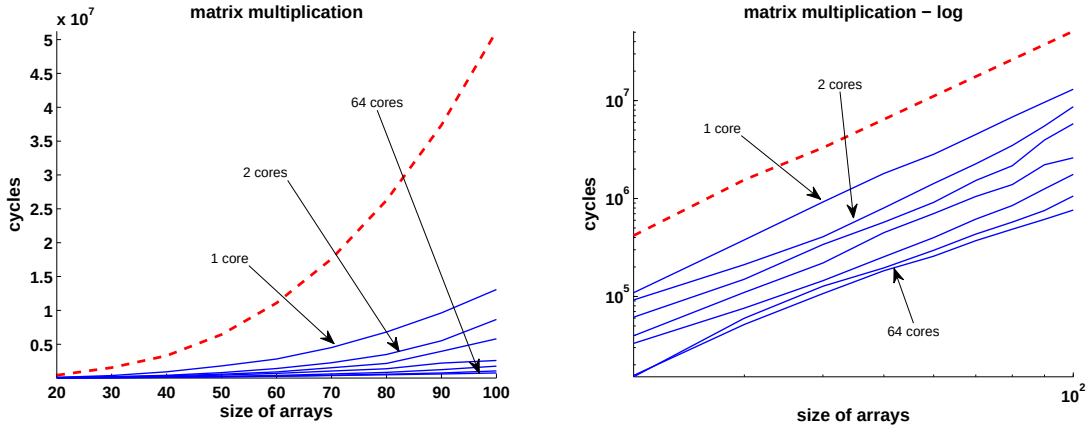


Figure 13: Matrix Multiplication. The dashed line is for the execution times for the serial version of the algorithm while the solid lines are for the automatically generated code running on SVP for 1,2,4,8,16,32 and 64 cores. The figure on the left presents the results in linear axis while the figure on the left is a log to log plot presenting the same information.

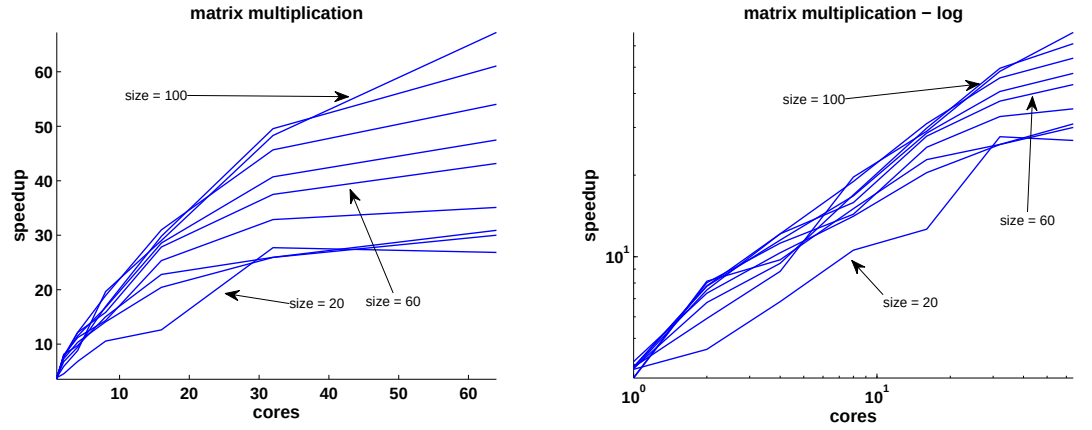


Figure 14: Matrix Multiplication. The speedup achieved when running on SVP for 1,2,4,8,16,32 and 64 cores. The figure on the left presents the results in linear axis while the figure on the left is a log to log plot presenting the same information.

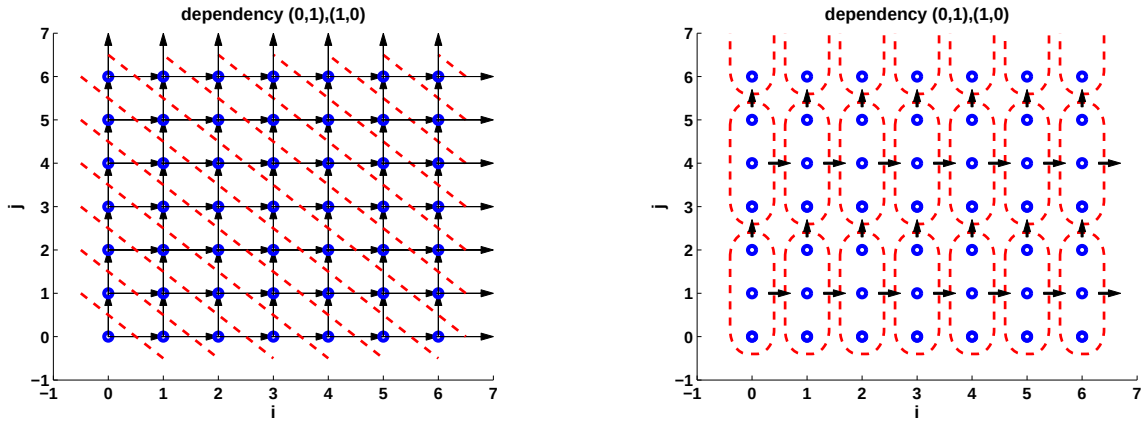
This is a two single dependency loop in a two dimensional array. Each element $A_{i,j}$ of the array depends on the elements $A_{i-1,j}$ and $A_{i,j-1}$ as shown in figure 15(a). Families of threads are created and scheduled according to the hyperplane model as discussed in the previous sections and shown in figure 15(b).

The speedup for different values of the size of the array ranging from $N = 100$ to $N = 1000$ and for different number of cores can be found in figure 16, in 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 many cores the execution on SVP is always faster and the trend implies even faster execution for larger values of N.

12.3 Experiments with Recursive Functions

We will present two experiments with well known problems. Experimental results have been collected using a hardware simulator of the SVP model [4] and for 8 cores.

In the first experiment we compute the numbers of the moves necessary to solve the Hanoi tower problem with N disks. The recursive code that solves the problem follows:



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

Figure 15: A loop with two dependencies: (0,1),(1,0)

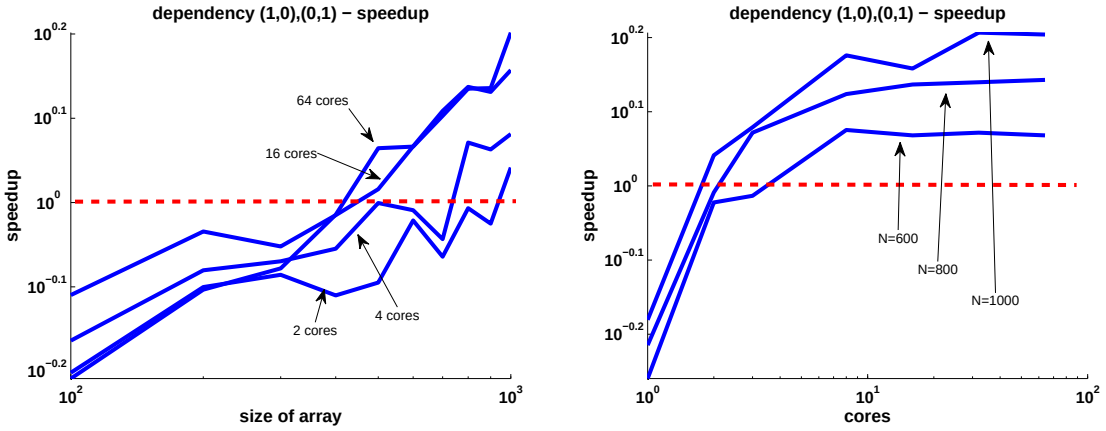


Figure 16: A loop with two dependencies: (0,1),(1,0). Speedup versus the size of the array and the number of cores. The dashed line indicates where the speedup is equal to 1.

```
long int hanoi(int x)
{
    if (x<0) return(-1); // not defined
    else if (x==0) return(0);
    else return(2*hanoi(x-1)+1);
}
```

In order to compute `hanoi(x)` the function `hanoi(x-1)` has to have finished the computation. The problem is serial, at least in its current form. A family of N threads will be created, all initializations will take place in parallel, all evaluation of conditions will also be executed in parallel and only the final computation in every thread will be serial. Thus, we expect an acceleration of the execution due to this parallelism. Execution times for the hanoi problem and for different number of disks is shown on figure 17.

Next we considered the fibonacci numbers problem. The code is listed in the previous section. Even though `fib(x-1)` and `fib(x-2)` can be invoked in parallel, there is replication of work which results in low execution times, even if we run it in parallel. We avoid the replication of work by moving two amounts from thread to thread, shifting the value of the first to the second and computing only the first amount. This leads to a significant speedup even if compare with the

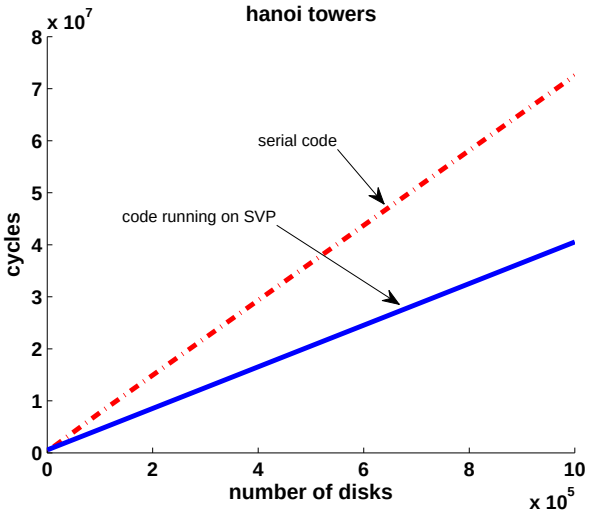
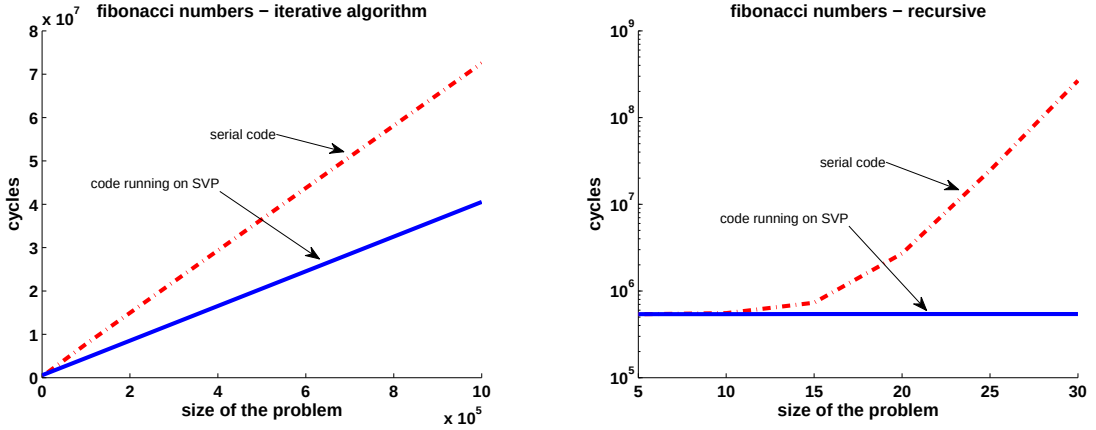


Figure 17: Execution times for Hanoi towers.



(a) Iterative implementation of fibonacci numbers.

(b) Recursive implementation of fibonacci numbers. Y-axis is in logarithmic scale.

Figure 18: Execution times for Fibonacci numbers.

iterative implementation of fibonacci numbers. The execution times for the iterative implementation is shown on figure 18(a) and the execution times for the recursive implementation is shown on figure 18(b). Please note that the recursive implementation can compute only the first few fibonacci numbers, and even in such a small problem size the execution time explodes. The reason that the line for SVP in figure 18(b) seems flat is that the y-axis is in logarithmic scale.

Please also note that the recursive hanoi and the iterative fibonacci numbers presents similar execution times. This is due to the optimizations performed by the gcc compiler which eliminates the recursion. The execution times on SVP is still better. In the fibonacci problem (i) gcc fails to perform optimizations and (ii) there is much replication of work, something that explains why the acceleration is that large.

12.4 Selecting the Optimal Number of Threads in a Family

The number of threads in a family is an important parameter for the $C2\mu TC/SL$ compiler which affects the performance of the produced application and unfortunately it can not be decided automatically. In this section we will perform some experiments on the number of threads in a family and we will select the optimal number of threads in a family for two loops with classic dependencies.

We first performed experiments for the two dimensional nested loop with two unary dependencies (1,0) and (0,1) the code of which and how the iterations are scheduled have been discussed in section

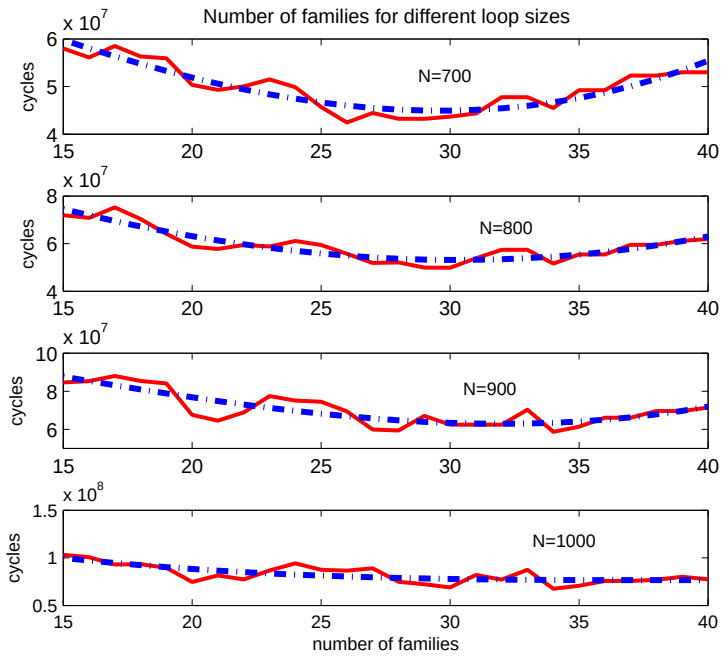


Figure 19: Machine cycles for different number of families and different loop sizes.

8. We studied the performance of the application for different sizes of the loop and for different number of families. The target was to find a relation between the size of the problem and the optimal number of families.

Experimental results are presented in figure 19 showing the performance of the execution of the nested loop on SVP for loop sizes ranging from 700 to 1000 and for the number of families ranging from 1 to 40. For large families (small number of families) the performance of the application is not good, something expected of course, since large families limits the parallelism in this specific application. For example, if the number of families is 1, then in order a second family start execution, the first one must have completed its own execution. The performance is improved as the number of the families increases (and their size becomes smaller) and until the execution time reaches to a minimum value. After this value the execution time increases again. The size of the family for which the minimum execution time is achieved is considered as the optimal number of threads in the family for the specific size of the problem.

In figure 19 machine cycles for different number of families and different loop sizes are depicted. Solid line is the real experimental results while the dashed line expresses the trend line computed using least squares approximation and a polynomial of 2nd degree. From this figure it can be easily seen that the optimal number of families for loop size equal to 700 is 26, the optimal number for size equal to 800 is around 30, the optimal number for size equal to 900 is 34 and the optimal number for size equal to 1000 is around 40. There is an almost linear relation between the size of the nested loop and the optimal number of families. The larger the size of the loop is, the larger number of families is necessary. For the specific experiments it seems that the optimal number of threads in a family is around 26-27 threads.

Next we present the speedup achieved for using the optimal size of families for the different sizes of the loop. The speedup is depicted in figure 20. Again the lines presented are the approximated line using least squares since it was easier to make observations on them. The results are similar to those of figure 19. For loop size equal to 700 the maximum value of the speedup is reached for 28 families, for loop size equal to 800 the optimal number of families is 30, for loop size equal to 900 the number of families becomes 33 and for loop size equal to 1000 it becomes 36. The optimal number of threads in a family ranges now from 25 to 27.

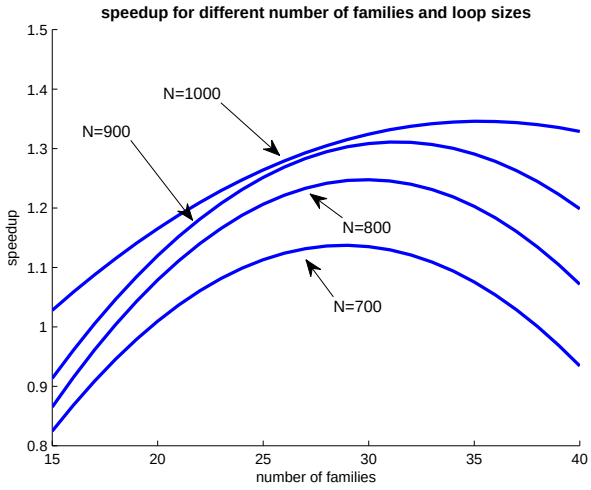


Figure 20: Speedup for different number of families and different loop sizes, D=(1,0),(0,1).

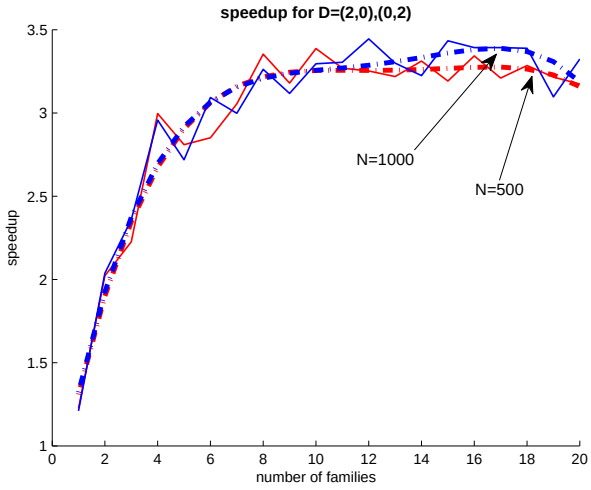


Figure 21: Speedup for different number of families and different loop sizes, D=(2,0),(0,2).

In the second experiment we double the size of each dependency and examine nested loop with dependencies (0,2),(2,0). Now there is more parallelism hidden in this loop. Since the dependency is 2, two families can start execution simultaneously and after these two complete execution all families depended on these families can start execution. The speed achieved is larger now as expected. The experimental results are depicted in figure 21 for loop sizes of 500 and 1000 iterations. Both curves behave in a similar way and reach to the optimal number of families early. The optimal number of families is between 15 and 20 for this problem. Again solid lines are the real experimental results and dashes lines represents trends approximated with the least square method and a polynomial of 6th degree.

12.5 Comparison with other Active Parallelizing Compilers

12.5.1 Similar work

During this project we downloaded, installed and deeply investigated Cetus [11–13] compiler infrastructure. We studied the literature, the available documentation and scanned the source code. Cetus includes a data flow framework, array section analysis, symbolic range analysis, and data dependence analysis. Basic techniques like privatization, reduction variable recognition and induction variable substitution, pointer analysis etc. are also supported. Cetus supports data dependence

analysis but still very limited. It seems that the code for data dependence analysis is still under development or at least not documented yet. The output file produced from Cetus does not contain any information related to data dependence analysis. However we found code in the source code of Cetus which we can assume that is for data dependence analysis. The data dependence analysis developed for $C2\mu TC/SL$ in this project is more completed, at least in those parts that could be used for extracting parallelism and mapping it on SVP. We used the version 1.2.1 of Cetus which was the latest available during the AppleCore project. In June 2011, after the end of the project, version 1.3 of Cetus was released. We have not investigated this version, we plan to do it soon.

Another interesting parallelizing tool is Pluto [24]. Pluto uses the polyhedral model for representing dependencies and transformations. Pluto automatically generates OpenMP code for multicore processors and optimizes sequences of imperfectly nested loops through tiling transformations. In [25] a compiler built on top of Pluto achieves better load balancing by optimizing at run-time the task graph. Even though C2mTC/SL has much in common with the above parallelizing compilers, it is at the same time quite different. C2mTC/SL (i) uses runtime scheduling and scheduling decisions are taken during the execution of the parallel application, (ii) groups iterations and schedules families of threads and (iii) targets a parallel processing model which supports both shared and synchronizing memory.

Other interesting approached which should be cited include Oscar [26,27], Polaris [28,29], Nanos [30], Paradigm [31,32] and VFC [33]. These compilers support input programs written in Fortran. Loops in Fortran and in Pascal are better structured than those in C, making much easier their analysis and the automatic parallelization. Some other parallelizing compilers, like Cetus, based on Polaris, SUIF [34], Parafrase–2 [35] and Promis [36], support input in C. Almost all of these compilers are based on the shared memory model. VFC supports message passing for the synchronization between processors. However, none of the available compilers support the memory model of the SVP (designed with both shared and synchronizing memory), something that makes the requirements of the problem and the design of the compiler different and interesting.

12.5.2 Unibench

In this subsection we will present some graphs generated by Unibench [37], a software tool developed by one of the partners for systematic evaluation and experimental analysis. The same experiments for which execution times are presented here have been performed for the SaC [7] compiler and are available in another deliverable. The graphs presented in figure 22 are for Livermore kernels 1,7,12 and 22.

12.5.3 Commercial Compilers

Commercial compilers focus on identifying segments of code which can execute in parallel. They are much closer to the targets of Cetus rather than those of $C2\mu TC/SL$. The reasons must be searched in the fact that commercial architectures provide hardware supporting fully parallel execution of code and not execution of code with dependencies. The support from hardware of execution of code with dependencies as supported by SVP is disruptive technology which is a strong point for AppleCore project and all tools developed in it. The different targets between commercial compilers and $C2\mu TC/SL$ makes $C2\mu TC/SL$ different from all commercial compilers and is something that characterizes this work as novel.

13 The Impact of the Workpackage

13.1 ... to the Research Community

Two papers describing part of this work have been presented in two specialized workshops. The first paper was presented in the workshop *Compilers for Parallel Computing* and described the design of the compiler and the specifications of the loop transformations. The second paper was accepted for presentation in the workshop for *Parallel Programming and Run–Time Management Techniques for*

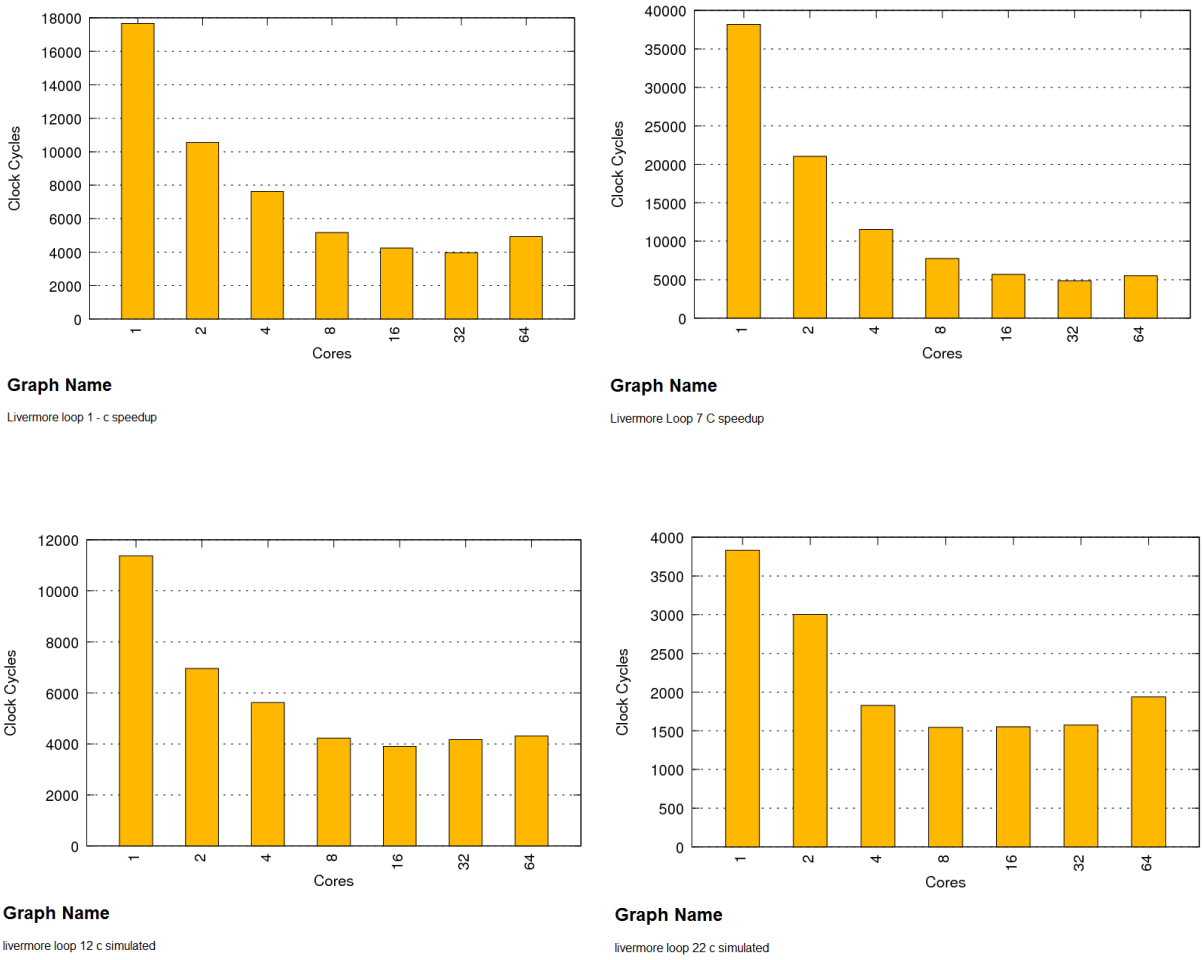


Figure 22: Clock cycles for Livermore kernels 1,7,12,22.

Many-Core Architectures, co-located with *ARCS (Architecture of Computing Systems) Conference* . In this paper we described the run-time scheduling mechanism of $C2\mu TC/SL$ and presented some evaluation results on the overhead produced by the run time scheduler. The complete references for those papers follow:

- Dimitris Saougkos, Despina Evgenidou and George Manis, "Specifying Loop Transformations for $C2\mu TC$ Source–to–Source Compiler," in Proc. of the 14th Workshop on Compilers for Parallel Computing, Zurich, Switzerland, January 2009

- Dimitris Saougkos and George Manis, "Run–Time Scheduling with the $C2\mu TC/SL$ Parallelizing Compiler," 2nd Workshop on Parallel Programming and Run–Time Management Techniques for Many-Core Architectures, in the Workshop Proceedings of Conference on Architecture of Computing Systems (ARCS 2011), pp. 151-157, Como, Italy, February 2011

Both papers received encouraging reviews and the presentation during the conferences was successful and the discussion followed constructive.

Another one paper has been accepted for presentation and will be presented in September in *Workshop on Language-Based Parallel Programming Models* co-located with the *9th International Conference on Parallel Processing and Applied Mathematics*. The subject of the paper is the extraction of parallelism from recursive functions. The complete reference follows:

- Dimitris Saougkos, Aristeidis Mastoras and George Manis, "Fine Grained Parallelism in Recursive Function Calls", in Proc of the Workshop on Language-Based Parallel Programming Models, organized during the 9th International Conference on Parallel Processing and Applied Mathematics, Torun, Poland, September 2011

13.2 ... to the Business Sector

One of the main purposes of the project was the development of the $C2\mu TC/SL$ compiler to influence the business sector. Within the scope of the Apple-CORE project, ACE made four main contributions.

From the start of the project ACE have extended and productized the loop carried dependence engine in CoSy. Loop carried dependence analysis can be used now in a future release of $C2\mu TC/SL$ to analyze standard loops in C to be automatically converted to micro-threaded C. This engine is now part of CoSy and it plays a central role in the analysis of loops for advanced transformations, including automatic parallelization.

ACE has extended its front-end technology with generic, scope sensitive, pragma based extensions to C. For CoSy, the novelty of this is that the syntactic scope of certain constructs, such as loops, can be maintained in the Intermediate Representation. This is not commonly done in the IR representation of the control-flow graph. It is useful because it enables the implementation of C extensions such as micro-threaded C by CoSy engines, instead of the front-end itself. This enhances the modularity of building target specific language extensions in CoSy. This mechanism was used in particular for the implementation of OpenMP front-end extensions, which are now part of the CoSy product.

Furthermore, ACE adapted its optimizing engines to support the SVP architecture's relaxed memory consistency model. This is relevant for any shared memory programming model. Even though the consistency model is relaxed (as opposed to "sequential consistency", which can be implemented in CoSy by treating all shared variables as volatile) it does not permit certain well-known optimizations that are valid for sequential programs. To be precise, relaxed consistency does not allow the compiler to introduce shared memory writes where the program does not have such writes. This happens in particular in optimizations that apply caching of global variables. The effort of this task went into checking existing engines for such occurrences of caching. It is made available in the CoSy product as an option to turn off caching in specific engines.

In the final part of the project, ACE concentrated its efforts on the implementation of a polyhedral solver. By itself, this allows a refinement of the loop carried dependence analysis mentioned above. Its novelty lies in the capability to compute and represent dependencies exactly, instead of conservatively as is commonly done in compilers. This opens new roads for program transformation in particular in the area of manipulating parallel programs. Although a huge step was made by the completion of the solver, this work is not yet part of the CoSy product. ACE intends to continue this effort to make it so.

14 Limitations

A number of limitations are listed in the following. The main reasons for those limitations is the nature of C which is a language with a much more free grammar than all other procedural languages. We list those limitations for completeness. Actually these limitations prohibits sometimes the automatic parallelization but a small effort for the restructuring of the code by the programmer is usually enough to overcome these difficulties.

- the program should contain only a `main()` function (other functions are ignored and not propagated to the generated code)
- the `main` function must contain at least one loop (otherwise the compiler rejects the program)
- all variables accessed by `main()` must be declared locally to the function (otherwise the compiler rejects the program)
- variables must have either primitive type or array type (no pointers or struct, otherwise the compiler either rejects the program or generates invalid code)
- the control flow cannot contain function calls (otherwise the compiler rejects the program)

- the loop indices must be integers (otherwise the bounds are rounded)
- the dependencies which can be detected are only static
- the command `goto` is not supported

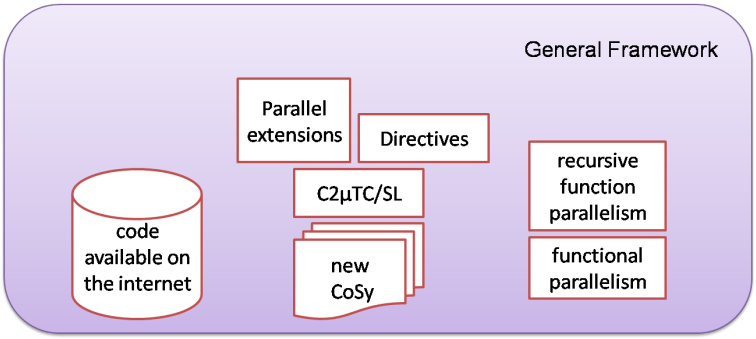


Figure 23: A more general parallelization framework.

15 Conclusion and Future Work

At the end of this project we believe that workpackage III achieved the initial targets and the software delivered showed that the automatic mapping of C programs onto SVP is feasible and efficient.

Now, the first release of the compiler is ready and publicly available. An extension of this work seems to be interesting. We plan to invest in several directions:

- exploit the publicly available code on the Internet from which we plan to extract supplementary information for parallelization
- extent the parallelism of recursive functions to a more general framework which can include general task parallelism
- add directives to the compiler as “hints” for both loop based and task parallelism in the cases which the compiler fails to extract parallelism due to complicated or not well written code, or in the cases in which the difficulty of automatically extracting parallelism is inherent.

The ultimate target is to develop a general framework which will combine all the software developed until today and extend it toward all the above directions as shown in figure 23.

References

[1] Dimitris Saougkos, Despina Evgenidou, and George Manis. Specifying loop transformations for $C2\mu TC$ source–to–source compiler. In *Proc. of 14th Workshop on Compilers on Parallel Computers*, Zurich, Switzerland, January 2009.

[2] Dimtris Saougkos and George Manis. Run–time scheduling with the C2μTC/SL parallelizing compiler. In *Proc. of 2nd Workshop on Parallel Programming and Run–Time Management Techniques for Many-core Architectures*, Como, Italy, 2011.

[3] 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.

[4] 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.

[5] Chris Jesshope, Mike Lankamp, and Li Zhang. The implementation of an SVP many-core processor and the evaluation of its memory architecture. *SIGARCH Comput. Archit. News*, 37:38–45, July 2009.

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

[7] Sven Bodo Scholz. Single assignment c – efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

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

[9] Multiple authors. *The SL language reference*. Computer Systems Architecture Group, University of Amsterdam, May 2011.

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

[11] Sang Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *Proc. of Languages and Compilers for Parallel Computing, 16th International Workshop*, pages 539–553, College Station, TX, 2003.

[12] Troy A. Johnson, Sang Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel P. Midkiff. Experiences in using Cetus for source-to-source transformations. In *Proc. of Languages and Compilers for Parallel Computing, 17th International Workshop*, pages 1–14, Irvive, CA, 2004.

[13] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42:36–42, 2009.

[14] Aristeidis Mastoras. Automatic parallelization of programs using Cetus. BSc Thesis, University of Ioannina, Dept. of Computer Science, June 2011. in Greek.

[15] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. Automatic parallelization of recursive procedures. *Int. J. Parallel Program.*, 28(6):537–562, 2000.

[16] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 72–83, 1999.

[17] R. L. Collins, B. Vellore, and L. P. Carloni. Recursion-driven parallel code generation for multi-core platforms. In *Design, Automation and Test in Europe (DATE)*, 2010.

[18] Akimasa Morihata and Kiminori Matsuzaki. Automatic parallelization of recursive functions using quantifier elimination. In *10th International Symposium on Functional and Logic Programming*, Japan, 2010.

[19] W. L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *LISP and Symbolic Computation*, 1(1):35–47, 1990.

[20] Joonseon Ahn and Taisook Han. An analytical method for parallelization of recursive functions. *Parallel Processing Letters*, pages 87–98, 2000.

[21] Dimtris Saougkos, Aristeidis Mastoras, and George Manis. Fine grained parallelism in recursive function calls. In *Proc. of Workshop on Language-Based Parallel Programming Models*, Torun, Poland, 2011.

[22] 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.

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

[24] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Programming Language Design and Inpmementations (PLDI)*, Arizona, USA, 2008.

[25] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. *SIGPLAN Not.*, 44:219–228, February 2009.

[26] Hironori Kasahara, Motoki Obata, Kazuhisa Ishizaka, Keiji Kimura, Hiroki Kaminaga, Hirofumi Nakano, Kouhei Nagasawa, Akiko Murai, Hiroki Itagaki, and Jun Shirako. Multigrain automatic parallelization in japanese millennium project IT21 advanced parallelizing compiler. In *PARELEC '02: Proc. of the International Conference on Parallel Computing in Electrical Engineering*, pages 105–111, Warsaw, Poland, 2002.

[27] Kazuhisa Ishizaka, Takamichi Miyamoto, Jun Shirako, Motoki Obata, Keiji Kimura, and Hironori Kasahara. Performance of OSCAR multigrain parallelizing compiler on SMP servers. In *Proc. of Languages and Compilers for High Performance Computing, 17th International Workshop*, pages 319–331, West Lafayette, IN, 2004.

[28] Keith A. Faigin, Stephen A. Weatherford, Jay P. Hoeflinger, David A. Padua, and Paul M. Petersen. The Polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, 1994.

[29] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[30] Marc Gonzalez, Eduard Ayguade, Xavier Martorell, Jesus Labarta, Nacho Navarro, and Jose Oliver. NanosCompiler: Supporting flexible multilevel parallelism in OpenMP. *Concurrency: Practice and Experience. Special issue on OpenMP*, 12(12):1205–1218, October 2000.

[31] P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[32] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the Paradigm compiler for distributed-memory multicomputers. In *Proc. of the ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[33] S. Benkner, K. Sanjari, V. Sipkova, and B. Velkov. Parallelizing irregular applications with the Vienna HPF+ compiler VFC. In *Proc. of High Performance Computing and Networking*, volume 1401, pages 816–827, Amsterdam, The Netherlands, 1998.

[34] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[35] Constantine D. Polychronopoulos, Miliand B. Gikar, Mohammad R. Haghighat, Chia L. Lee, Bruce P. Leung, and Dale A. Schouten. The structure of parafrase-2: an advanced parallelizing compiler for C and FORTRAN. In *Selected papers of the 2nd Workshop on Languages and Compilers for Parallel Computing*, pages 423–453, 1990.

[36] Hideki Saito, Nicholas Stavrakos, Steven Carroll, Constantine D. Polychronopoulos, and Alexandru Nicolau. The design of the PROMIS compiler. In *Computational Complexity*, pages 214–228, 1999.

[37] Daniel Rolls, Carl Joslin, and Sven-Bodo Scholz. Unibench: A tool for automated and collaborative benchmarking. In *18th IEEE International Conference on Program Comprehension*, June 2010.