apple core

SEVENTH FRAMEWORK PROGRAMME

FP7-215216

Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs

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

Report on Code Transformation from C to μTC

Deliverable D3.2, Issue 1.2

Workpackage WP3

Author(s):	George Manis		
Reviewer(s):	Chris Jesshope, Clemens Grelck		
WP/Task No.:	WP3	**Number of pages:**	17
Issue date:	2008/12/10	**Dissemination level:**	Public

Purpose: Describe code transformations for $C2\mu TC$ compiler
Results: A set of code transformations focusing on loop structures
Conclusion: Sufficient parallelism can be extracted from loop structures when targeting the μTC language

Approved by the project coordinator: Yes **Date of delivery to the EC:** 2008/12/12

Document history

When	Who	Comments
2008/11/05	George Manis	Initial version
2008/11/12	George Manis	Comments by C. Jesshope and C. Grelck
2008/12/09	George Manis	Minor corrections

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

Table of Contents

1 Introduction **1**

2 Terminology and Definitions **1**

3 Basic Loop Transformations and Code Optimizations **2**

4 Transformations on Simple Loops **6**

5 Transformations on Nested Loops **10**

6 Summary **16**

1 Introduction

This document aims to present the basic code transformations which are to be implemented in the $C2\mu TC$ compiler. Parallelisation to the μTC language involves identifying both data or loop concurrency as well as concurrency between calling and called functions. Since most of the parallelization will be extracted from loop structures, this report focuses on loops. Parallelisation of the functional concurrency is in any case relatively straightforward from the user language perspective, the issues and problems occur in the generation of assembly code and its attendant restrictions.

A paper based on the main ideas presented in this deliverable has been accepted in CPC09 workshop (14th Workshop on Compilers for Parallel Computers 2009) and will be presented in Zurich in January 2009 [1].

The implementation will be done using the CoSy[1] [2] compiler development system. The source language will be the C programming language and the target the μTC [3, 4] programming language.

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

2 Terminology and Definitions

In this section we will define the terms *dependency* and *anti–dependency*, we will show how we depict iterations, dependencies and anti–dependencies in figures and we will give definitions for the terms *dependency vector*, *uniform*, *single* and *unary dependency*, and for the *iteration space*.

A statement S_2 is *flow–dependent* or *true–dependent* or simpler *dependent* on the statement S_1 if and only if S_1 modifies a variable (or generally uses a resource) that S_2 reads, S_1 precedes S_2 in execution and there is no statement in the control flow between S_1 and S_2 that reassigns this variable. The following is an example of a flow dependence, since S_1 must assign 1 to `x` before S_2 read it:

```
S1: x:=1;
S2: y=x+1;
```

A statement S_2 is *anti–dependent* on the statement S_1 if and only if S_2 modifies a variable (or generally uses a resource) that S_1 reads, S_1 precedes S_2 in execution and there is no statement in the control flow between S_1 and S_2 that reassigns this variable. The following is an example of an anti-dependence, since S_1 must use the variable `y` before S_2 modify it:

```
S1: x:=y+1;
S2: y=1;
```

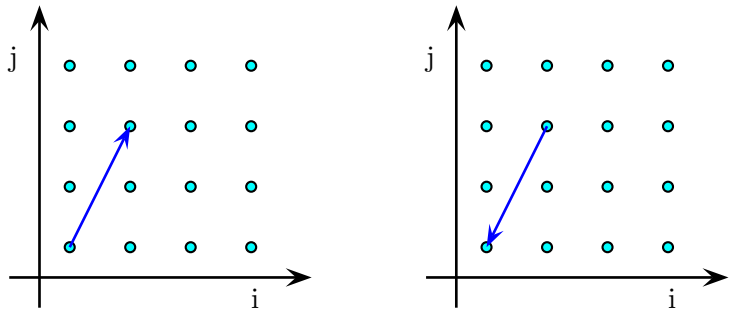
Anti-dependencies are not real dependencies. They are the result of using the same place in memory. They can always be eliminated by using additional variables. The above code is equivalent with the following one where the anti-dependency has been eliminated by replacing the variable `y` in S_2 and in the code that follows S_2 with `t`:

```
S1: x:=y+1;
S2: t=1;
```

In figure 1(a) circles represent loop iterations for the two dimensional loop:

```
for (i=1;i<N;i++)
  for (j=2;j<N;j++)
    x[i][j]=x[i-1][j-2];
```

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



(a) The arrow represents the dependency (1,2), i.e. one iteration back for i and two for j

(b) The arrow represents the antidependency (-1,-2), i.e. one iteration forward for i and two for j

Figure 1: Representation of dependencies and anti–dependencies in figures. Circles represent loop iterations. Axes show the loop indices.

The horizontal axis shows the values of the loop variable i and the vertical one the corresponding values for j. The arrow represents the dependency (1,2), i.e. one iteration back for i and two for j. According to the previous definitions this is a flow–dependency.

In the case of the anti–dependency (-1,-2), i.e for the loop:

```
for (i=0;i<N-1;i++)
  for (j=0;j<N-2;j++)
    x[i][j]=x[i+1][j+2];
```

the figure is exactly the same but the arrow has the opposite direction, as shown in figure 1(b). Only flow dependencies are real dependencies. Anti–dependencies are usually introduced by the programmers in order to write more efficient code or save some memory. In a parallel processing environment the anti–dependencies may result into unnecessary locks in memory and unnecessary delays. Fortunately we can always eliminate an anti–dependency by replicating information.

The *dependency vector* between iteration $\vec{i} = (i_1, i_2, \ldots, i_n)$ and iteration $\vec{j} = (j_1, j_2, \ldots, j_n)$, when $\vec{i}$ depends on $\vec{j}$, is the vector $\vec{d} = \vec{i} - \vec{j} = (j_1 - i_1, j_2 - i_2, \ldots, j_n - i_n)$.

A dependency is *uniform* when the dependency vector is constant.

A dependency is *unary* when all the elements of the dependency vector are equal to 1 or 0.

A dependency is *single*, when the dependency vector consists of one element.

The *iteration space* $I \in \Re^n$ is the set of the values for the loop indices vector $\vec{i} = (i_1, i_2, \ldots, i_n)$, as specified by the loop limits.

3 Basic Loop Transformations and Code Optimizations

Loops have concentrated the attention of many researchers trying to extract parallelism from sequential programs since most of the execution time is usually consumed inside loops, sometimes exceeding the 90% of the total execution time. However, the extraction of parallelism is not an easy task and the problem still remains open.

A survey on loop transformations for high performance computing has been published in “ACM Computing Surveys” [5] concentrating most of the experience of this research community in this subject. Another similar paper published has been published earlier in “Communications of the ACM” describing compiler optimizations for supercomputers [6]. Since then, many of papers have been published in the same field, proving that the problem remains open. Most of these loop transformations are widely used by commercial and research compilers. The most important are listed in the following. Some of them are already supported by CoSy and will be used as they are, some other are necessary or at least very useful for the development of a parallelizing compiler with the philosophy of $C2\mu TC$, and some others are just listed for completeness and will be reconsidered in later stages of the development if they can help us or not. Loop transformations include:

◇ *Loop–based strength reduction* replaces an expression with an equivalent one but less expensive, e.g.:

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

becomes:

```
t=c;
for (i=0;i<N;i++)
{   a[i] += t;
    t+=c;
    }
```

Strength reduction is supported by CoSy.

◇ In *Loop–invariant code motion*, when the result of a computation does not change between iterations, the compiler can move the computation outside of the loop, e.g.:

```
for (i=0;i<N;i++)
    a[i] += sqrt(x);
```

becomes:

```
if (n>0) t=sqrt(x);
for (i=0;i<N;i++)
{   a[i] += t;
    t+=c;
    }
```

Loop–invariant code motion is supported by CoSy and will be used as it is.

◇ *Induction variable elimination:* with the term *induction variable* we mean a variable whose value is derived from the index. Sometimes some of these variables can be eliminated. CoSy supports induction variable elimination.

◇ *Loop interchange* exchanges the position of the outer and the inner loops in a nested loop, e.g.:

```
for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        a[i][j] += c;
```

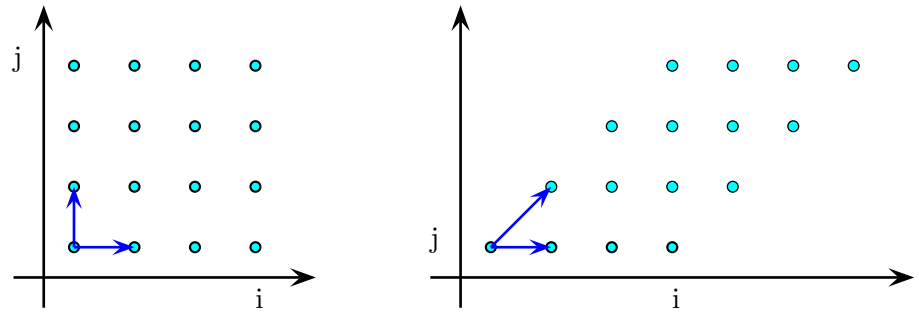
becomes:

```
for (j=0;j<N;j++)
    for (i=0;i<N;i++)
        a[i][j] += c;
```

Loop interchange might help us identifying parallelism in conjunction with other transformations.

◇ *Loop skewing* adds the outer loop index multiplied by a skew factor f to the bounds of the inner loop index and then subtracts the same quantity from every use of the index of the inner loop inside the loop body. Skewing does not change the meaning of the program and is always legal. It transforms the iteration space and can change its shape from a rectangle to a parallelogram or vice versa.

Figure 2 illustrates how the iteration space is modified. In the iteration space on the left, the dependency vectors are(1,0) and (0,1). The iteration space is skewed by a factor $f = 1$ and becomes as shown on the right hand side of the figure 2. Now the dependency vectors are (0,1) and (1,1) something that allows parallelism. Skewing is not supported by CoSy, it is useful for the projects and will be implemented now.



(a) The iteration space before skewing

(b) The iteration space after skewing

Figure 2: With loop skewing the iteration space is modifies in order to increase parallelism. In this figure the iteration space is skewed by the factor $f = 1$.

◇ In *Loop reversal* the iterations are executed in the opposite direction, i.e. the index is runs from the upper limit to the lower limit. For example:

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

becomes:

```
for (i=N;i>=0;i--)
    a[i]++;
```

Loop reversal can eliminate the use of temporary variables. CoSy supports loop reversal.

◇ *Strip Mining* transforms a serial loop into an outer loop and an inner parallel loop. The index of the outer loop increases with a predefined step and divides the index space into blocks of a predefined size. The index of the inner loop steps through each one of those block. For example:

```
for (i=0;i<N;i++)
    a[i] = a[i] + c;
```

becomes:

```
for (j=0;j<N;j+=block)
    for (i=j;i<j+block;i++)
        a[i] = a[i] + c;
```

Strip mining is very useful for our project. Our interest is to extract independent loops as these can be arbitrarily distributed but also to produce loops with a single uniform, unary dependency, since these are efficiently handled by our hardware exploiting do–across schedules. Strip mining is not supported by CoSy and will be implemented now as part of the $C2\mu TC$ compiler. Strip mining can also help us identify loops in operations efficiently implemented by our processor, e.g. reduction.

◇ *Loop tiling* is a generalization of strip mining into more than one dimensions. Again, for the same reasons mentioned in strip mining, loop tiling is a very useful transformation for our project.

◇ *Cycle shrinking* is a specialization of strip mining, since it transforms a serial loop into an outer loop and an inner parallel loop exploiting that the dependency is greater than 1, e.g.:

```
for (i=2;i<N;i++)
    a[i] = a[i-2] + c;
```

becomes:

```
for (j=2;j<=3;j++)
    for (i=2;i<N;i+=2)
        a[i] = a[i-2] + c;
```

With cycle shrinking we can again produce loops with a single uniform, unary dependency which can run in parallel. There is no support from CoSy for such kind of analysis and transformation and will be implemented for the purposes of appleCore project.

◇ *Loop splitting* breaks a loop into many and *Loop fusion* does exactly the opposite.

```
for (i=2;i<N;i++)
{   a[i] = a[i-1] + c;
    b[i] = b[i-2] + c;
    }
```

becomes two loops when loop splitting is applied:

```
for (i=2;i<N;i++)
    a[i] = a[i-1] + c;
for (i=2;i<N;i++)
    b[i] = b[i-2] + c;
```

and can become again one with loop fusion

Loop splitting can help us extract parallelism. In the above example from the version with the two `for` we can extract more parallelism since (i) the first loop can be executed by the hardware of our processor and (ii) in the second loop we can detect cycle shrinking. Loop splitting is not supported by CoSy. Loop distribution is supported by CoSy.

◇ In *Loop normalization* the index starts always from 0 (or 1) and the step is always 1. It may be used complementary to other transformations and simplifies the development of the code. CoSy supports loop normalization. An example of loop normalization follows:

```
for (i=2;i<N;i++)
    a[i]=a[i-1]+1;
```

becomes:

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

Other transformations like loop unrolling, software pipelining, loop coalescing, loop collapsing, loop peeling, loop spreading, etc, do not seem to be useful for the development of $C2\mu TC$ compiler. In later stages of development we will reconsider if they can help us or not.

There are also other code transformations–optimizations which can be applied in order to make a program run faster and these include: constant propagation, constant folding, copy propagation, forward substitution, algebraic simplification, unreachable code elimination, useless code elimination, dead variable elimination, common sub-expression elimination, short-circuiting, procedure elimination, or procedure call elimination, parameter promotion, procedure cloning, etc. These transformations are more or less common knowledge for the compiler development community and most of them are already implemented as optimizations in the CoSy framework. We will exploit all the already available engines of CoSy to perform code optimization to the maximum possible degree.

In the following sections we will discuss how we will apply these transformations and produce μTC code from serial C and what other transformations are necessary. First we will discuss for one–dimensional loops and later for higher dimensions. The transformations discussed in these sections are customized for μTC and our processor.

4 Transformations on Simple Loops

In the simplest case there is a single unary dependency between the elements of an array. This kind of dependency is supported directly from the hardware of the processor. The transformation is straightforward as shown in the following code:

Code in C:

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

Code in μTC:

```
thread body(shared int s, int * w)
{   index i;
    w[i]=s+1;
    s=w[i];
    }

thread main(void)
{   int fid;
    int s;
    ...
    s=w[0];
    create(fid;;1;N-1;1;;;)
        body(s,w);
    sync(fid);
    }
```

The threads generated by the `create` construct is a *family of threads*. The i_{th} element of the array w depends on the $(i-1)_{th}$ element of the same array. The $(i-1)_{th}$ element must be computed before the i_{th} element is computed. The value of the $(i-1)_{th}$ element will be passed from the $(i-1)_{th}$ thread (iteration) to the i_{th} thread (iteration) through the shared variable s. The value of s is initialized to the value of $w[0]$ in the main thread. In every other thread (iteration) the value passed by the previous thread (iteration) through s is received with the statement `w[i]=s+1` and written in the main memory. The next statement allows the $(i+1)_{th}$ thread (iteration) to receive the value of `s=w[i]`.

Suppose now we have an anti–dependency, where the i_{th} element depends on the $(i+x)_{th}$, where of course $x > 0$ since we have an anti–dependency. The $(i+x)_{th}$ iteration can execute only when the i_{th} iteration is completed. This is a severe limitation for the performance of the loop. Since anti–dependencies are not real dependencies, we can always write equivalent code which does not contain anti–dependencies.

In the following example we copy the array w to a new temporary array t and then allow all iterations to run independently, using array t for reading and w for writing. The code in C and its μTC equivalent follows:

Code in C:

```
for (i=0;i<N-x;i++)
  w[i]=w[i+x]+1;
```

Code in μTC:

```
thread body1(int * w, int * t)
{   index i;
    t[i]=w[i];
    }

thread body2(int * w, int * t)
{    index i;
     w[i]=t[i+x]+1;
    }
```

```
thread main(void)
{   int fid1,fid2;
    int t[N];
    ...
    create(fid1;;0;N-1;1;;;)
        body1(w,t);
    sync(fid1);
    create(fid2;;0;N-x-1;1;;;)
        body2(w,t);
    sync(fid2);
    }
```

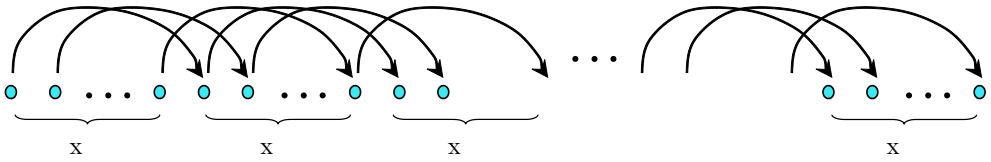


Figure 3: A uniform dependency with distance x can be transformed to x unary uniform dependencies. All x loops can execute in parallel.

If we have a single dependency, but not a unary one then the loop is transformed from a single dimension loop to a loop of a higher dimension. It is equivalent with x independent loops, where x is the dependency distance, the first one starting from x the second starting from $x+1$ and the last from $2x-1$. Each one has a step equal to x, as shown in figure 3. The code in C and μTC follows.

Code in C:

```
for (i=x;i<N;i++)
    w[i]=w[i-x]+1;
```

Code in μTC:

```
thread body2(shared int s, int * w)
{   index j;
    w[j]=s+1;
    s=w[j];
    }

thread body1(int * w)
{   index i;
    int fid,s;
    s=w[i];
    create(fid;;i;N-1;x;;;)
        body2(s,w);
    sync(fid);
    }

thread main(void)
{   int fid;
    ...
    create(fid;;x;2*x-1;1;;;)
        body1(w);
    sync(fid);
    }
```

Suppose now we have more than one dependencies ($x > 0$):

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 $(i-1)$, in a way presented in figure 4. The code follows ($x > 0$):

Code in C:

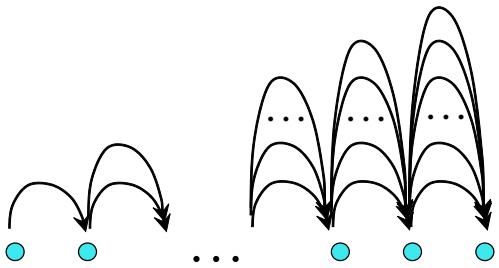


Figure 4: Data produced in iterations $i-2, i-3, \ldots, i-x$ move from iteration to iteration in order to reach the i_{th} iteration where they will be used.

```
for (i=x;i<N;i++)
  w[i]= w[i-1]+w[i-2]+...+w[i-x]+1;
```

Code in μTC:

```
thread body(shared int s_0, s_1, ..., s_{x-1},int *w)
{   index i;
    w[i]=s_0 + s_1 + ... + s_{x-1};
    s_0 = s_1;
    s_1 = s_2;
    ...
    s_{x-1}=w[i];
    }

thread main(void)
{  int fid,s_0, s_1, ..., s_{x-1};
    s_0=w[0];
    s_1=w[1];;
    ...
    s_{x-1}=w[x-1];
    create(fid;;x,N-1,1;;;)
        body(s_0, s_1, ..., s_{x-1},w);
    sync(fid);
    }
```

This code, however, needs some explanation. Every iteration uses x shared variables. The first iteration (where $i = x$) initializes these variables with the first x values of the array and computes the element $w[x]$. Then, the shared variables are shifted by one position (the content of the shared variable s_0 is lost) and assigns the element $w[i]$ to the variable s_{x-1}. The second (where $i = x + 1$) iteration, and generally the iteration with index i (where $i \neq x$) reads the shared variables from the previous iteration, calculates the element $w[i]$ using all x shared variables and prepares the shared variables for the next iteration in a way similar to that described earlier.

Finally, when there are both dependencies and anti–dependencies we move the anti–dependencies before the dependencies, we categorize them according to the above examined cases and handle the problem as a combination of these cases. An example follows (again $x > 0$):

Code in C:

```
for (i=1;i<N-x;i++)
  w[i]=w[i-1]+w[i+x]+1;
```

Code in μTC:

```
thread body1(int * w, int * t)
{   index i;
    t[i]=w[i];
    }

thread body2(shared int s, int * w,int * t)
{   index i;
```

```
    w[i]=s+t[i+x]+1;
    s=w[i];
    }

thread main(void)
{   int fid;
    int t[N];
    int s;
    ...
    s=w[0];
    create(fid;;0;N-1;1;;;)
        body1(w,t);
    sync(fid);
    create(fid;;0;N-x-1;1;;;)
        body2(s,w,t);
    sync(fid);
    }
```

Some reduction operations can efficiently be computed on our processor. In *reduction* we compute a scalar value from an array. Typical examples are the computation of the sum or the maximum element of an array. Some reductions are easy to be identified and transformed into μTC. As an example, let us consider the following summation:

```
sum=0;
for (i=0;i<N;i++)
    sum+=w[i];
```

becomes:

```
thread body(shared int sum, int * w)
{    index i;
     sum+=w[i];
    }

thread main(void)
{   int fid;
    int sum=0;
    ...
    create(fid;;0;N-1;1;;;)
        body(sum,w);
        sync(fid);
    }
```

However, we can do much better than that if we use a technique similar to strip mining (please see section 3). Two or more families of threads can be generated, each one computing part of the total sum. After all families terminate, the total sum is computed as the summation of all partial sums:

```
thread body(shared int sum, int * w)
{    index i;
     sum+=w[i];
    }

thread main(void)
{   int fid;
    int sum;
    int sum1=0,sum2=0;
    ...
    create(fid1;;0;(N-1)/2;1;;;)
        body(sum1,w);
    create(fid2;;(N-1)/2+1;N-1;1;;;)
        body(sum2,w);
```

```
    sync(fid2);
    sync(fid1);
    sum=sum1+sum2;
    }
```

5 Transformations on Nested Loops

The transformations of nested loops are not just a straightforward generalization of simple loops. The processor supports communication between shared variables of successive threads (iterations) only. In one–dimensional loops the data produced in iterations $i-2, i-3, \ldots, i-x$ can be transferred to the i_{th} iteration trough the iteration $i-1$. However this is not always possible, or at least not always efficient in nested loops, something that makes the problem more complicated and more interesting. We will start from the simpler cases where generalizations from simple loops are possible.

The problem with the anti–dependencies is again simple. A copy of the array is done exactly in the same way as in one dimensional loops. Here the generalization is straightforward. If there are both dependencies and anti–dependencies, we move all the anti–dependencies before the dependencies, we produce code which handles the anti–dependency, we synchronize and then produce the code for the dependency.

As an example we give the following code, where $a > 0$ and $b > 0$. A copy t of the array w is created and every loop iteration reads from t.

Code in C:

```
for (i=0; i<N-a; i++)
    for (j=0;j<N-b;j++)
        w[i][j]=w[i+a][j+b];
```

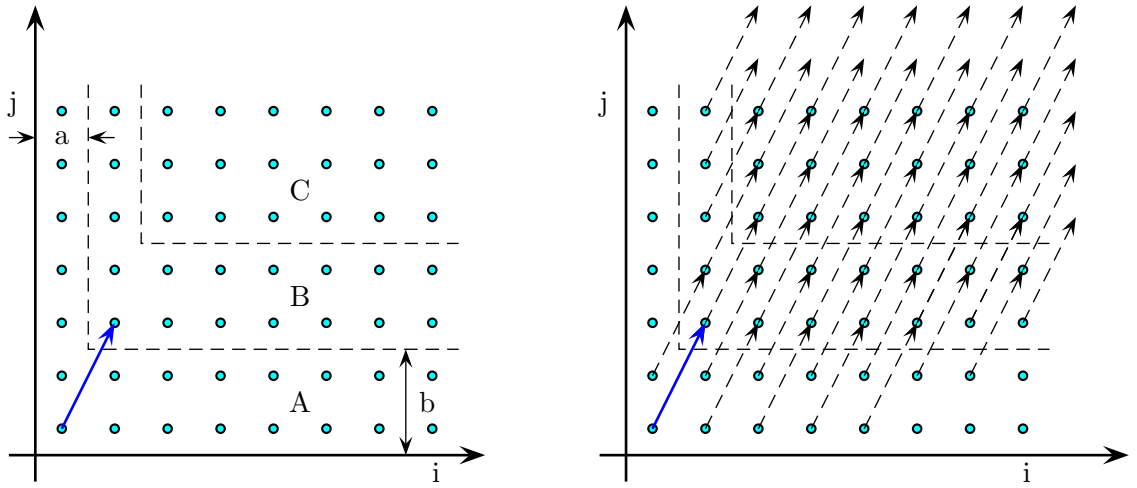
Code in μTC:

```
thread body2(int i, int a, int b, int ** t, int ** w)
{   index j;
    t[i][j]=w[i+a][j+b];
    }

thread body1(int a, int b, int ** t, int ** w)
{   index i;
    int fid;
    s=w[i];
    create(fid;;i;N-b-1;1;;;)
        body2(i,a,b,t,w);
    sync(fid);
    }

thread main(void)
{   int fid;
    ...
    create(fid;;0;N-a-1;1;;;)
        body2(a,b,t,w);
    sync(fid);
    }
```

of what we did in simple loops. Suppose the following C code:

```
for (i=0; i<N-a; i++)
  for (j=0;j<N-b;j++)
    w[i][j]=w[i-a][j-b];
```

In the case now in which there is only one uniform dependency the generalization is still possible. We will generate loops which can execute in parallel. These loops will be one–dimensional, each one starting from a different point, but all of them having the same step. In figure 5(a), i and j range from 1 to N. The space is divided into three distinct subspaces. The first one (subspace A)



(a) The space is divided into three subspaces. Points in B depend on points in A. Points in A are not updated by the loop.

(b) Parallel loops start from every point in B and execute independently.

Figure 5: Independent loops exploiting full parallelism for single dependencies

consists of those values of i and j which are not part of the iteration space ($i < a,\ j < b$). The second subspace (subspace B) consists of all pairs of i and j which depend on the pairs in subspace A. The number of pairs in B is equal to $N(a+b) - 3ab$. The rest of the iteration space is marked as subspace C. An one–dimensional loop corresponds to each one of the point in subspace B. As shown in figure 5(b) each one of the $N(a+b) - 3ab$ loops starts from one of the points in B and continues the execution independent from all other loops.

The μTC code is shown in the following ($a > 0,\ b > 0$):

```
thread t1(int a0, int b0, int a, int b, int ** w,
    shared int ia, shared int ib, shared int s)
{   index i;
    ia+=a;
    ib+=b;
    w[ia][ib]=s;
    s=w[ia][ib];
    }

thread t2(int a0, int a, int b, int ** w)
{   index b0;
    int ia=a0;
    int ib=b0;
    int s=w[a0-a][b0-b];
    int fid3;
    create (fid3, a, N-1, 1)
        t1(a0, b0, a, b, w, ia, ib, s);
    sync(fid3);
    }

thread t2'(int a0, int a, int b, int ** w)
{   index b0;
    int ia=a0;
    int ib=b0;
    int s=w[a0-a][b0-b];
    int fid3;
    create (fid3, a, 2a-1, 1)
        t1(a0, b0, a, b, w, ia, ib, s);
    sync(fid3);
    }
```

```
thread t3(int a0, int a, int b, int int ** w)
{   int fid2;
    create (fid2, b, 2b-1, 1)
        t2(a0, a, b, w);
    sync (fid2);
    }

thread t4(int a0, int a, int b, int ** w)
{   int fid2;
    create (fid2, 2b, N-1, 1)
        t2'(a0, a, b, w);
    sync (fid2);
    }

thread tp1(int a, int b, int ** w)
{   index a0;
    t3(a0,a,b,w);
    }

thread tp2(int a, int b, int ** w)
{   index a0;
    t4(a0,a,b,w);
    }

thread main()
{   int fid1,fid2;
    ...
    create (fid1;;a,N-1,1;;;)
        tp1(a,b,w);
    create (fid2;;a,2*a-1,1;;;)
        tp2(a,b,w);
    sync(fid2);
    sync(fid1);
}
```

Please note that this transformation is faster than the hyperplane model [7] where a wavefront of computation is produced, i.e. iterations which can execute in parallel. We produce here parallel families of threads exploiting the synchronizing memory and the rapid flow od data between successive iterations in the same family. The family of threads resembles to a pipeline of execution.

Let us now have a look at some more difficult case. Not every code transformation of nested loops can derive from what was discussed in section 4. 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, but we look for something better, a customized solution for our processor.

The processor supports single unary dependencies. We will select one of the dependencies and produce a family of threads for this. The rest of the dependencies fall in one of the following categories.

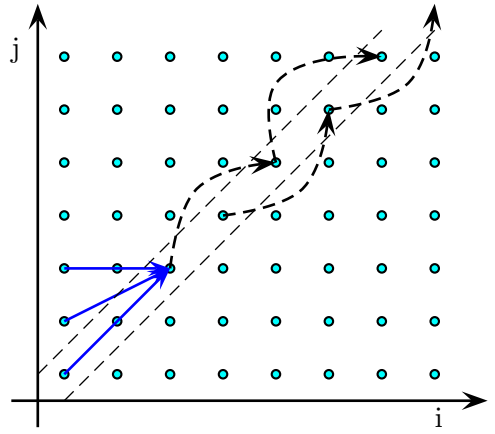
- ◇ if the iteration i depends on an iteration j which belongs in the same family of threads, the data produced by iteration j will move from iteration to iteration until they reach to iteration i.

- ◇ if the iteration i depends on an iteration j which belongs in a different family of threads, the execution of which has been completed earlier, it will read the data produced by the iteration j from the main memory. Since the `create` construct corresponding to the family of threads

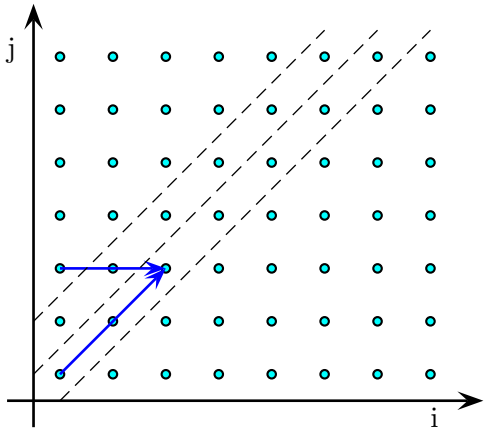
in which the iteration j belongs to is completed, the memory is considered to be consistent and all future computations can read this data.

Anti–dependencies are handled in a similar way.

- ◇ if the iteration i is related with iteration j which belongs in the same family of threads with an anti–dependency, a copy t of the array w is created and every thread in the family reads the values of w from t.
- ◇ if the iteration i belongs in the family of thread I and depends with an anti–dependency relation on the iteration j which belong in the family of threads J, then we can assume that the family I will complete the execution before the family J starts the execution. Thus, each iteration i can read from the main memory the values of j, since j will modify them after the family I completes the execution.



(a) Two families of threads run in parallel, one for the odd iterations and one for the even ones.



(b) The dependencies allow two families of threads, marked with dashed lines, running in parallel.

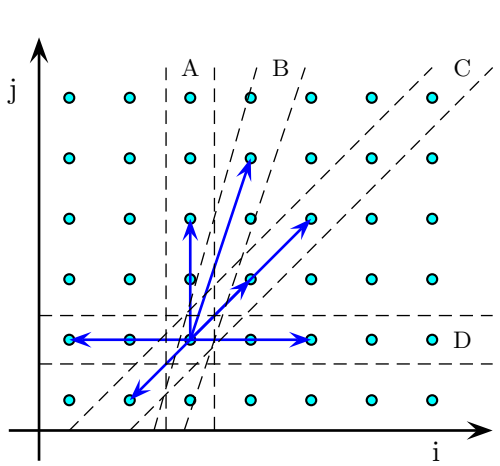
Figure 6: Independent loops exploiting full parallelism.

Other generalizations of simple loops transformations can also apply here. In the figure 6(a) an example is shown, where the dependency (2,2) allows the execution of two families of threads running in parallel, one for the odd iterations and one for the even ones. In a way similar to what was discussed for simple loops, the dependency (x,x) allows here x families of threads execute concurrently.

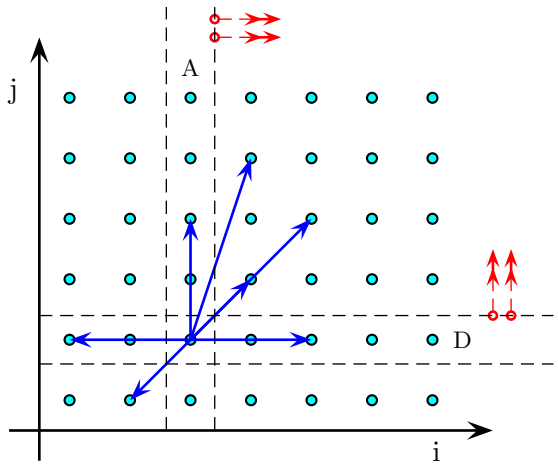
In figure 6(b) the values read from the memory due to the dependency (2,0) have been computed by a family of threads which has completed the execution earlier. Please note that these dependencies allow two families of threads run in parallel, each one executing iterations marked with the dashed lines in the figure.

In figure 6(b) we were lucky enough to select a dependency and create families of threads which produce the same results with the serial execution when these families of threads are created from the left to the right. However, we must be careful because this is not always the case. Please see figure 7(a). If we select to create our families based on the dependency (1,1) something that seems to be a reasonable choice (two dependencies and one anti–dependency are covered by the same family of threads) the results produced are not correct. When the point (i,j) is to be calculated the points (i-2,j) and (i,j-2) should have been already calculated. However, this is not possible, no matter if we create families of threads from the left to the right or from the right to the left. Both pipelines B and C are not legal and cannot be selected. On the other side, pipelines A and D is possible to produce correct results. If we move towards the directions shown with dashed arrows in figure 7(b) all dependencies fall on elements calculated by families of threads which have already completed their executions.

Let us consider once again the following example:



(a) Families of threads are created based on the dependencies. However, not all dependencies produce the same results with the serial execution.

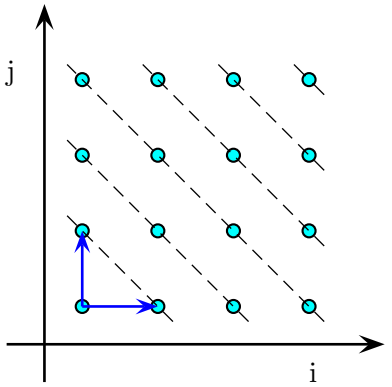


(b) Pipelines A and D produce correct results when families of threads are created in the direction showed above.

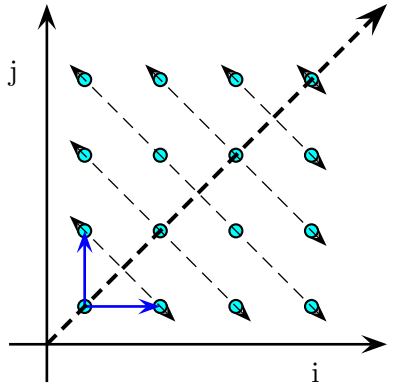
Figure 7: Legal execution, families of threads and dependencies

```
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 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 [7] we can find which points can be computed in parallel. In figure 8(a) iterations on each one of the dashed lines can run in parallel.



(a) The dependencies (1,0),(0,1) do not allow parallelization. However, iterations on each dashed line can execute in parallel



(b) A nested create which exploits parallelism. The dark dashed arrow shows the outer loop, the dashed parallel arrows shows the inner loop

Figure 8: The hyperplane model

We would like to customize the hyperplane method for our target architecture. The transformations presented earlier can cot capture this concurrency. If we select one of the two dependencies as the main dependency, as we did in figure 7(b), all we can get is a serial execution. Without loss of generality we select the dependency (1,0) and assign each line of the iteration space to a separate family of threads. However, since the memory is considered coherent only after the execution of the family of threads is over, two of these families cannot run in parallel. The only benefit is the fast transfer of data from one iteration to the other through the synchronizing memory.

A solution to this problem is a nested `create` in which the outer loop runs all iterations lying on the main diagonal. For each one of these iteration the inner loop runs all iterations lying on the dashed line crossing the diagonal at this iteration (see figure 8(b)).

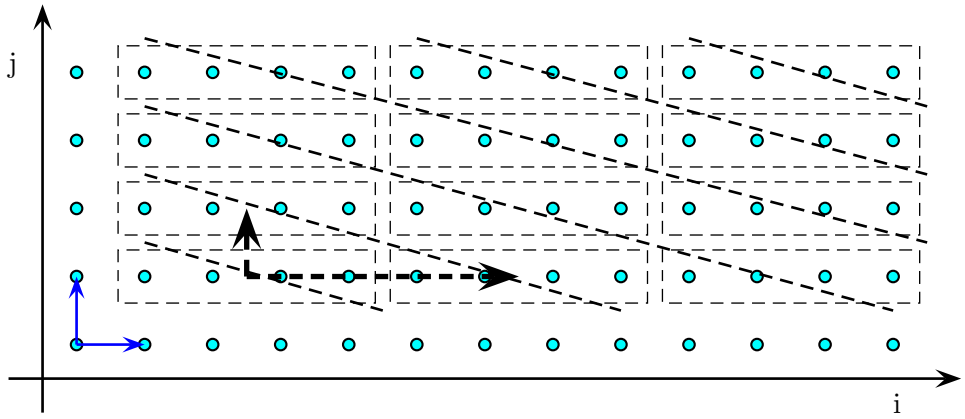


Figure 9: Dependencies between iterations are reduced into dependencies between families of threads. Parallel execution of families of threads according to the hyperplane model is possible and increases hardware utilization

However, this solution has an important drawback. It is a very good approach for a conventional parallel processing model since all iterations in the inner loop are independent. This leads to full parallelization and high processor utilization. Here we have interesting in solutions which exploit the special characteristics of our target processor. This solution does not exploit the synchronizing memory to the maximum degree. A better solution, which actually is a customization of the hyperplane model to our processor is shown in figure 9.

The challenge here is to find a solution which allows more than one family 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 8(b). Dashed boxes are families of threads. Upon finishing execution, a family of thread can launch the family on the right hand side and on the top of it as shown by the dark dashed arrow. In this way more than one family can execute in parallel. We exploit both the parallelism resulting from the hyperplane model theory and the underlying hardware, since we have families of threads exploiting the synchronizing memory. The problem is reduced from a problem with dependencies between iterations to a problem with dependencies between groups of successive iterations, i.e. families of threads.

Please note here that it is important, or at least useful, that the dependency we select (as we did on figure 7(b)) is convenient to be on the horizontal (or vertical) axis (as it is one figure 9). This is not a problem indeed. In section 3 we saw how we can use loop skewing to transform the iteration space in an equivalent one. We can use loop skewing in order to transform the iteration space so that one of the dependencies is on the horizontal (or vertical) axis.

An issue related to multiple dependencies is that there is a limitation on the number of the shared registers which can be used in hardware level. At the moment, such kind of low-level restrictions like this are not known to us. Our strategy is to ignore these restrictions now and postpone these problems for later stages of the project where the design and implementation of the processor will be more complete. Even though this is not subject of this deliverable, we wanted at least to identify the potential problem.

6 Summary

In this document we present the basic code transformations for translating C code into μTC. Since we expect to extract most or all of the parallelism from loop structures, we focus on loops. We study the available loop transformations in the literature and decide which ones are useful to us. We also present in detail the loop transformations as we plan to implement them in the development of $C2\mu TC$.

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

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

[4] C.R. Jesshope. SVP and μTC– A dynamic model of concurrency and its implementation as a compiler target. Internal Report, Computer Systems Architecture Group, University of Amsterdam, The Netherlands, 2007. http://staff.science.uva.nl/ jesshope/papers.html.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[6] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12), 1986.

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