FP7-215216

Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs

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

Specification of Changes Required to the CoSy Framework

Deliverable D3.1, Issue 2.0

Workpackage WP3

Author(s):	George Manis, Marcel Beemster		
Reviewer(s):	Chris Jesshope, Raphael Poss		
WP/Task No.:	WP3	**Number of pages:**	13
Issue date:	2008/12/10	**Dissemination level:**	Public

Purpose: Describe modifications for the *CoSy* framework
Results: A set of modifications-extensions to *CoSy* engines
Conclusion: *CoSy* is flexible enough to facilitate the development of $C2\mu TC$ compiler. A small number of extensions-modifications is required

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

Document history

When	Who	Comments
2008/10/20	George Manis	Initial version
2008/10/31	George Manis	Incorporation of ACE's comments
2008/11/01	George Manis	Text (section 4) from ACE's document added
2008/11/09	George Manis	Comments by C. Jesshope and R. Poss
2008/12/08	George Manis	Process model added (section 6) from ACE's document

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

Table of Contents

1 Introduction **1**

2 The *CoSy* Compiler Development System **1**

3 The μTC Language **3**

4 From C to the Micro-Threaded Architecture **4**

5 Focus on selected Engines and necessary Modifications **6**

6 CoSy's Process Model **7**
6.1 The CoSy Intermediate Representation 8
6.2 The CCMIR Process Model Requirements 8
6.3 The Shared Memory Consistency Model 9
6.3.1 Example 1: the HW/SW Interface 9
6.3.2 Example 2: a Compiler Optimization 9
6.3.3 Example 3: Dealing with Flush 10
6.4 Memory Consistency Models for CoSy 11
6.4.1 Zero Consistency 11
6.4.2 Sequential Consistency 11
6.4.3 Relaxed Consistency 12

7 Summary **12**

1 Introduction

This document aims to present the extensions necessary to be done in the *CoSy* [**?**] compiler development system in order to facilitate the development of $C2\mu TC$ [**?**], a parallelizing source-to-source compiler targeting the μTC [**?**, **?**] 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 The *CoSy* Compiler Development System

CoSy[1] is a professional, easy-targetable and highly flexible compiler development system from ACE Associated Compiler Experts, which has been successfully deployed by over 100 industrial customers and partners world-wide, in creating high-quality, high-performance compilers for a broad spectrum of targets running from 8-bit microcontrollers to CISC, RISC, DSP and 256-bit VLIW processor architectures.

Based upon its highly modular design surrounding a generic, extensible intermediate representation (IR) and extensive use of generators, the *CoSy* environment enables construction of production-quality performance compilers in a highly efficient manner, reducing time-to-market, time-to-performance, as well as development and maintenance costs.

CoSy is currently used by:

- ◇ Many semiconductor companies for fastand cost-effective development of production-quality compilers for new processor architectures
- ◇ Development tool and EDA companies for development of commercial software development tools
- ◇ Architecture R&D groups for performance testing and architecture roadmap exploration
- ◇ Academia and Research Institutions for research on compilation techniques and processor architecture design

In *CoSy*, optimal code selectors and optimization strategies are generated from descriptions that reflect the features, parallelism and timing of the architecture. And as the focus in producing compilers is moved to creation of the processor description, optimizing compilers can be available as soon as architecture specifications are stable. Compilers built with the *CoSy* compiler development system are inherently of high quality and performance. To reach this level, front-ends, analysis algorithms, optimization algorithms, code selectors and register allocators are all engineered and generated as self-contained engines. These engines perform their specific function on the IR in co-operation with all the other engines configured into the compiler. Typically there will be over fifty of these independent engines in a compiler. A generated compiler supervisor controls the order in which the engines are invoked, and their interactions. Particularly with respect to the latest powerful processor architectures, the dynamics of compiler optimization can be quite challenging. During the compilation process many decisions are made in various phases of the compiler. Such decisions may seem appropriate in the context of a particular phase, but may be suboptimal, or even counter productive from the perspective of subsequent phases. With *CoSy*'s independent engines, interacting on the IR and managed by the supervisor, the phase ordering challenge is more easy to grasp, as the basis is provided for powerful performance and space optimization strategies.

A wide range of analysis and optimization techniques are available incorporated within various engines. Below find the most important ones:

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

Analysis techniques:

- ◇ Advanced loop analysis
- ◇ Alias analysis
- ◇ Available expressions analysis
- ◇ Code use-estimate analysis & prediction
- ◇ Data flow analysis
- ◇ Def-use analysis
- ◇ Dominator tree analysis
- ◇ Leaf procedure analysis
- ◇ Optimized code debug support
- ◇ Profiling: Path (dynamic) profiling & static profiling
- ◇ Value range analysis

Optimization techniques:

- ◇ Algebraic optimization
- ◇ Base binding
- ◇ Basic block reordering
- ◇ Branch optimization
- ◇ Code simplification
- ◇ Common sub expression elimination
- ◇ Common tail merging
- ◇ Constant sharing: Floating Point constant sharing & String sharing
- ◇ Constant folding
- ◇ Constant optimization
- ◇ Constant propagation
- ◇ Control flow optimizations: Chain flow optimization, If simplification, Straightening
- ◇ Copy propagation
- ◇ Data size optimization
- ◇ Dead code removal: Unreachable code, Unused code
- ◇ Dead object removal: Dead function elimination, Dead variable elimination
- ◇ Delay-slot filling
- ◇ Expression canonization
- ◇ Forward substitution
- ◇ Function inlining
- ◇ Hardware loop generation
- ◇ Instruction combining
- ◇ Instruction scheduling: Hyperblock Forward & backward list Look-ahead & exhaustive
- ◇ Leaf procedure optimization
- ◇ Life range splitting

- ◇ Loop optimizations: Loop canonization, Loop fusion, Loop hoisting, Loop induction variable rewriting & elimination, Loop invariant code motion, Loop inversion, Loop scalar replacement, Loop removal, Loop reverse conversion, Loop unrolling, Software pipelining
- ◇ Normalization
- ◇ Operator overloading
- ◇ Post-increment optimizations
- ◇ Predicated execution
- ◇ Register coalescing
- ◇ Register promotion
- ◇ Scalar replacement of aggregates and array references
- ◇ SIMD instruction generation
- ◇ Strength reduction
- ◇ Switch optimization
- ◇ Tail recursion elimination
- ◇ User-defined intrinsic functions

3 The μTC Language

μTC has been designed as an intermediate language to support concurrency in a range of computing architectures from conventional distributed systems through to reconfigurable platforms. In particular, in relation to the Apple-CORE project, it supports the programming of many-core chip multiprocessors (MCCMPs). μTC was originally defined in the AETHER EU project as an abstract concurrent processor called SVP (self-adaptive virtual processor), where the goal was to define a dynamic concurrency model able to support self-adaptive computing. It can be implemented in conventional software using OS or language support in existing systems or can be implemented in hardware at the level of the ISA in emerging MCCMPs. It is the latter that is the focus of the Apple-CORE project.

Self-adaptivity is the ability of a system to react to its environment in order to optimize its performance, which could be processing power, energy consumption or fault tolerance, all issues that will be critical in future silicon technologies. More importantly, a self-adaptive system is able to manage a large set of resources dynamically, rather than forcing a compiler to perform a static analysis and mapping. These properties make SVP and ideal programming environment for this new generation of many-core chip-multiprocessors.

The following properties are required in the abstraction to support self-adaptative systems:

- ◇ to capture all of a programs concurrency: this is required as scheduling concurrency is the only mechanism we have to support self-adaptation. We also note that sequencing a concurrent description is trivial when compared to extracting concurrency from a sequential one;
- ◇ to capture locality of communication: this is required in order to reflect future constraints on communication in silicon systems; finally
- ◇ to keep everything as dynamic as possible in a dynamically changing environment and to support the widest range of applications possible.

Work on the SVP definition within AETHER and the corresponding C-based language that supports it, namely μTC, is now complete, although work continues within Apple-CORE in defining a more formal semantics of the language and the objects that it supports. This language will be used in Apple-CORE for developing a tool chain to program MCCMPs. The additions to C that define μTC and capture the SVP abstractions are defined below. These constructs are used to

dynamically create and identify a unit of work that is a parameterised family of threads. The basic type that supports this concept is the family identifier, which is new type added to the C language. An implementation of μTC will encode this to identify the family, its location and a capability to control it, the latter being a random number generated on family creation. The constructs defined below allow the creation and termination of identified families of threads, which may be defined hierarchically. The concurrency captured at any stage in the execution of the program is then defined by the tree of families with the original job or task at its root. μTC adds the following keywords to standard C. An informal syntax and semantics of each construct is given in the following sections. The new keywords are:

- ◇ *create*: Construct used to create a family of concurrent microthreads;
- ◇ *thread:* Type qualifier identifying functions as threads;
- ◇ *squeezable:* Function qualifier identifying threads that propagate a squeeze signal to subordinate families;
- ◇ *shared:* Type qualifier for variables of a type that will be shared between adjacent microthreads in a family;
- ◇ *index:* Type specifier for variables that will represent the index value of a thread;
- ◇ *sync:* Construct used to identify the termination of a specified family;
- ◇ *break:* Construct used to terminate the execution of a family from one of its threads;
- ◇ *squeeze:* Construct used to preempt the execution of a specified family so that it may be restarted without loss of state;
- ◇ *kill:* Construct used to terminate a specific family with a prejudice;
- ◇ *family:* Type specifier used to specify a variable that identifies a family of threads;
- ◇ *place:* Type specifier used to specify a variable that identifies a place at which to execute a family of threads.

4 From C to the Micro-Threaded Architecture

Given the following code written in C:

```
int i, t, a ;
t = 0 ;
a = 42 ;
for( i = 0 ; i < 100 ; i++ ) {
    t = some_work( t ) ;
    more_work( i, t, a ) ;
}
return t ;
```

Assume that compiler analysis can find that all calls to the function `more_work()` in this loop are independent of each other, implying that they can be executed in parallel. It is not obvious for a compiler to do such analysis, but that is not the topic of this paper.

Secondly, assume also that the computations in the function `some_work` are independent of the computations in the function `more_work`. This can also be possibly analysed by a compiler.

Our goal might be to parallelize the loop. However, the two assumptions are not strong enough to do so because there is a loop carried true dependence between subsequent iterations through the variable `t`. Every next iteration uses the value of `t` from the previous iteration.

It is however possible to pipeline the execution of subsequent loop iterations. The next iteration can start as soon as the value of `t` is know from the previous iteration. This is the strength of the micro-threaded architecture. It is capable to manage efficiently, in hardware, each subsequent iteration of the loop as a separate process. Additionally, it can set up communication between subsequent iteration threads to efficiently pass value such as `t` between them.

The task of the compiler to generate code for the micro-threaded architecture is to transform the loop into the creation of a family of (hardware) threads, where each thread executes one iteration and where each thread sets up the right environment.

From the compiler's perspective, the micro-threaded model defines:

- ◇ Thread families, where a family typically corresponds to a for-loop. Each thread in the family executes one iteration of the loop, so every thread in the family executes the same code. A specific register, defined by the micro-threaded model, can be used in the thread to find out which iteration it is.

- ◇ Global registers that are the same for every thread in the family. Global registers are read-only. They are typically used for variables, such as `a` in the example above, that are defined once before the loop.

- ◇ Read-only shared registers that are used to get values from the directly preceding iteration thread (or from the family creator in case of the first iteration). These communication registers have the special property that they will block execution of the thread until a value is written into it by the preceding iteration.

- ◇ Write-once shared registers that are used to transport values to the next iteration thread, where they are visible as read-only registers. For the last thread, the written to register can be recovered by the thread family creator. The combination of the read and write shared registers defines a synchronized communication channel between subsequent iterations. So, this limits communication to one direction only, in increasing iteration number. Also, communication is only possible to the directly subsequent iteration thread, it is not possible to make larger jumps.

- ◇ It is not possible to have any other communication between threads in a family beyond the shared registers interaction. This implies that there may not be updates (writes) to shared memory between threads. However, families of threads may communicate via shared memory providing the producer family has terminated and the consumer family can see that termination either in the same thread or possibly in a concurrent one using a synchronising shared variable. This restriction is necessary for the hardware to operate efficiently. It allows the hardware thread scheduler freedom to manage threads within the available resources and it allows the memory system to cache variable within threads without having to worry about shared memory consistency.

The micro-threaded model allows loops that only have loop carried dependencies with the next iteration to be executed with higher efficiency than sequential execution. The maximum speedup is limited by the critical path of the dependence through all threads. Each iteration contributes the time it takes to compute the new value from the previous value of the computation on the critical path. For the example, it is the time it takes to compute the new value of `t` with the function `some_work` from the old value of `t`. The compiler can contribute by scheduling the critical path as densely as possible. If there is more than one dependence only the longest single critical path contributes, at least if the hardware scheduling of threads is optimal.

If there are no loop carried dependencies all iterations can execute in parallel and there is no critical path that goes through all of them. The compiler can contribute by moving loop carried dependencies out of the inner loop through loop interchange transformations.

5 Focus on selected Engines and necessary Modifications

CoSy offers a number of engines which can be applied on its intermediate representation. The wide range of the available engines gives us the opportunity to apply several analysis and optimizations techniques, most of which are very useful or absolutely necessary for any compiler in order to obtain high quality of produced code and achieve the required performance.

CoSy is a general purpose compiler framework. The internal design and philosophy make it easy-targetable and highly flexible. We will exploit this characteristic in order to enrich *CoSy*'s engines to support automatic code parallelization, an application field with many special requirements.

Most of the already available engines will be used as they are and combined with each other to enable code optimization and analysis. Some other will be extended in order to fulfil the special requirements of the project. The rest of the code will also be written as a *CoSy* engine in order to be coherent with the overall philosophy and enforce code reusability.

Let us now have a look at those engines which are strongly related to this project. Since loops are very significant we start from loop engines:

- ◇ *Loopanalysis:*
  produces loop markers, i.e. identifies the index of the loop and the induction variables, the init block (the basic block which initializes the variables), the test block (the basic block which checks if the index exceeded the limit or not) and the body (the basic block which is actually the main body of the loop). This engine does not need any modifications or extensions. The output of the analysis can be used for μTC "create" structures construction. Thus, this is an important engine for the project since we expect to extract most of the parallelism from loop structures.

- ◇ *Loopinvariant:*
  moves loop invariant code outside the iterating loop body. It can be used for code optimization. This engine needs no modification or extension.

- ◇ *Loopremove:*
  finds and removes empty loops. It can also be used for code optimization. This engine needs no modification or extension.

- ◇ *Loopive:*
  reduces the number of walking loop variables. It can be used for code optimization. This engine needs no modification or extension.

Another very important aspect is the *CCMIR* intermediary language. As mentioned earlier, *CoSy* is based on an intermediate code representation produced from the input program and a set of engines performing on this intermediate code in order to analyze it, optimize it or produce the final code for the target architecture which is usually the assembly of a specific processor. The target of this project is slightly different than the usual target of a compiler. We want to produce a high level language, similar to the input language, enriched with some extensions to exploit the special characteristic of a specific multi-threaded processor. Thus we need to extend the *CCMIR* intermediary language; then we need to ensure that the existing *CoSy* engines can operate properly on the extended *CCMIR*; then we need to extend the *Pirtoc* engine to emit source code with the μTC extensions.

However, reproducing or even producing high level source code from a lower level representation is a difficult task and the produced code is usually far away from the expected result. The output usually resembles to a low-level programming language rather than code written by a programmer. Even though this is expected and well understood, further improvement of the engine to produce code much closer to the human way of thinking and program writing could be useful.

The *CCMIR FSDL* definitions are the parts of *CoSy* that need to be modified to support concurrency in the intermediate representation. *Pirtoc* is the engine which then needs to be extended

to support μTC constructs in the output language. The decision which is the best way to do that is not subject of this deliverable since it can be considered as part of the $C2\mu TC$ compiler implementation rather than an extension to *CoSy*. Other alternatives which involve processing the output of the engine and not modifying the engine will also be investigated.

A very important issue is related with array and the transformation of array indices into walking pointers. *CoSy* produces an intermediate representation from the high level programming language in which some of the high level information related to array indices is not retained. *CoSy* tries to retain as much as possible information, but it is also required to make a semantically correct mapping to the IR. Thus, a very reasonable approach is to decide to convert indices into walking pointer when the size of the array is decided at runtime. However, for a source-to-source compiler this information is important and should be preserved. More specifically:

- ◇ *CoSy* demotes arrays of unspecified size to pointers. This loses the information "this object is an array"
- ◇ *CoSy* transforms all subscript operations on pointers to pointer arithmetic. This loses the information "this expression involves this array and this subscript"
- ◇ *CoSy* transforms arrays of dependent sizes (where the size is determined by another variable) to pointers. This loses the size information entirely
- ◇ *CoSy* transforms multi-dimensional arrays of dependent sizes to pointers, flatting the array to one only dimension. This loses any size information and the dimension of the array.

Unfortunately, all these problems are actually due to the nature of the C programming language and not unlucky design choices of *CoSy*. The C language specifications mandates a compiler to do so. However, ideally, we would like to avoid all the above problems entirely. So we need to either extend the current front-end to keep array subscripts or write a post-processing engines to recover them right after the front-end. The first alternative is preferred.

ACE works on the development of a loop control dependence analysis. What has been implemented until now is a very interesting approach, not directly related to the requirements of the $C2\mu TC$ compiler though. For the purpose of the specific project, a similar engine should be implemented from scratch or the existing engine should be extended in order to support amongst other things:

- ◇ calculation of the dependence matrix (dependence vectors amongst iterations)
- ◇ list with all elements read and written inside the loop

Finally, we will exploit some of the available engines performing code optimizations. The main advantage of using an existing compiler framework rather than building a new one from scratch is the possibility to exploit all the experience and knowhow acquired from the design, development and maintenance as well as to re-use the code. *CoSy* has been designed and implemented in such a way that the application of selected engines is possible and easy. We will use all the engines that can perform optimizations and make our final program run faster. We mentioned in section 2 many optimization techniques supported by *CoSy*, most of them classic in compiler development technology and we intend to use them for the implementation of our compiler. Those engines need no modifications or extensions.

6 CoSy's Process Model

Until now we have discussed some necessary extensions which seem to be very useful for the development of $C2\mu TC$ compiler. In this section we will present CoSy's *Process Model*, an extension of CoSy inspired from appleCore project, which might be very useful for future extension of $C2\mu TC$.

For ACE, an important (commercial) requirement is that CoSy's process model must be suitable not only for micro-threaded architectures but also other, similar, process models such as

OpenMP and multi-threading. The general process model to support is that of shared memory multi-processing.

In this section we will sketch the process model requirements in the CoSy context, memory consistency requirements on compiler optimizations, the relation between the micro-threaded architecture abstraction and the process model, and several examples.

6.1 The CoSy Intermediate Representation

CoSy's intermediate representation is called CCMIR, for Common CoSy Medium-level Intermediate Representation. CCMIR is a well defined program representation that is source language and target architecture independent. It is used by many analysis and transformation engines in CoSy. *Engines* operate on the CCMIR.

In its current form of CoSy Release 2008, CCMIR is a sequential program representation. To extend CCMIR with calls for creating and starting processes within a single address space is relatively straightforward. The requirements for that are described in subsection 6.2. However, the fact that processes share memory, and thus potentially read and write to the same memory locations, requires a precise definition of the memory consistency model and its impact on optimizations. This is discussed in subsection 6.3.

Note that even though micro-threaded parallelism can only be used when iterations do not communicate through shared memory, it is necessary to explicitly define a memory consistency model in order to specify the limits of compiler optimizations. Examples of why this is necessary are presented below.

6.2 The CCMIR Process Model Requirements

It is expected that the CoSy Process Model (CPM) consists of a small extension to the CCMIR proper (the CoSy Intermediate Representation) and a number of support functions that are defined as CKFs. CKFs are Compiler Known Functions, which are also known as intrinsics. CKFs are a standard feature of CoSy that allows extension of the compiler without having to change the CCMIR itself.

Requirements on the CPM are:

- ◇ CPM must be source language and target architecture independent. This is more a goal than an requirement.
- ◇ CPM must be suitable for shared memory multi-threaded programming models such as micro-threaded-C, OpenMP and (UNIX') pthreads.
- ◇ CPM must make it possible to map to both low-overhead and high-overhead implementations. In a low-overhead (few clock cycles per thread creation) implementation such as the micro-threaded architecture, direct mapping to architecture registers is required to keep the software overhead on top of the hardware minimal.

  In a high overhead implementation (like OS level processes) the creation of processes requires expensive resource allocation. For such implementation, reuse of processes for different tasks is possibly an efficient strategy.
- ◇ A suitable memory consistency model must be defined and handled by compiler optimizations.
- ◇ CPM must be independent of the communication model. This is to decouple the CPM from the choice of communication model. Communication in this case does not include instantiation of arguments from parent to child and return values from the child, which is included in the CPM.

  The micro-threaded architecture specifies nearest neighbor communication between subsequent threads, which is actually closely related to the parent to child instantiation. CPM must be able to incorporate this form of communication as an extension.

6.3 The Shared Memory Consistency Model

In this text, the term *shared memory* is considered equivalent with *single address space*. Shared memory consistency models are necessary when two processes run simultaneously and use data that resides in the same memory space. Considering shared memory consistency is even important when processes are not allowed to communicate through shared variables, as is the case for the micro-threaded architecture. Memory consistency must be considered at the hardware and the system level (including the compiler). The following examples illustrate this.

6.3.1 Example 1: the HW/SW Interface

```
shared variable a, b ;
PROCESS 1: { a = a + 3 ;
           }
PROCESS 2: { b = b + 5 ;
           }
```

In Example 1, the two processes 1 and 2 can run in parallel. Obviously, these processes should not be able to interfere with each other. However they can in the following, not unlikely scenario. Suppose the variables a and b are `short` integer with a size of 16 bits. Also suppose that the native architecture word-size is 32 bits and that the architecture only supports 32 bit load and store operations. Finally, for reasons of data storage efficiency, suppose that the variables a and b are stored together in the same 32 bit word.

In this scenario, an atomic write to the variables a and b is not possible. To write to variable a, the compiler needs to generate code that 1) reads the whole 32 bit word that also contains b into a register; 2) updates the a-part of the register with the new value of a; 3) writes the register back to the 32 bit word.

Now suppose that during step 2) of the update of variable a the whole statement of Process 2 is executed. In that case, in step 3) of the update of a, the value of b is overwritten with an old and incorrect value.

From a memory consistency point of view, the error is introduced because Process 1 temporarily (non-atomically with respect to other processes) caches the value of b, which is an effect that is not specified in the program.

In this case, the error must be avoided by making sure that the compiler never places multiple shared objects in the same 32-bit word. Note also that similar cases occur in hardware cache implementations in particular in multi-processor implementations. Typically, cache lines are larger than the individual elements in it. The hardware must make sure that an update of a single element in a line is an atomic operation with respect to other observers (processes) of elements in the cache line.

6.3.2 Example 2: a Compiler Optimization

```
shared variable c, x ;
PROCESS 1: { if (c)
               x = x + 1 ;
           }
PROCESS 2: { for (...)
               if (not c)
                   x = x - 1 ;
           }
```

In this example, assume that neither of the two processes updates the variable c, both see the same value for that variable. With that given, either process 1 updates variable x, or process 2 updates variable x, but never both. According to the program, there is no interference between the two processes regarding variable x.

A valid optimization for a sequential program is to avoid reading and writing memory locations by temporarily caching global variables into registers. After this optimization, Process 2 may be transformed into the following sequentially equivalent code:

```
shared variable c, x ;
register variable r ;
PROCESS 2: { r = x ;
             for (...)
                if (not c)
                    r = r - 1 ;
             x = r ;
            }
```

Unfortunately, this optimization breaks memory consistency between the two processes. Suppose the variable c is TRUE. In the original code, only Process 1 would write to variable x. In the optimized code, both processes write to variable x. Since Process 2 caches the value of variable x, even though it does not modify it, it may overwrite the value of x that was changed by process 1 with the old value.

So, optimizations exist that are valid for sequential code but invalid for parallel code. It is therefore necessary to investigate the optimizations of the compiler for a micro-threaded architecture regarding their validity in a parallel setting.

From a memory consistency point of view the error is introduced because Process 2 temporarily caches a variable while this was not specified in the program. This was exactly the same problem as in Example 1, although the nature of the optimization was completely different.

6.3.3 Example 3: Dealing with Flush

In the previous examples there was no communication over shared variables between threads. This is relevant for the micro-threaded architecture as its programming model does not allow for such communication. From the examples it should be clear that even in that case the hardware and software have certain responsibilities toward a correct implementation.

The micro-threaded programming model does support shared memory communication between parent and child processes. This is perhaps obvious because parents and children do not execute at the same time (parents spawn children, then wait until they are all done). It is still necessary to be aware of this as parents and children do run in different processes. In particular, all writes to shared memory by the parent must be completed at a synchronization point right before the first child starts executing, and all writes by the children must be completed at the synchronization point that is right after the last child ended but before the parent resumes. At the multi-threaded hardware level this is more complicated to implement than in the compiler. In the compiler this requirement is very similar to function calling (to an unknown function) in a sequential environment.

Within a single micro-thread, between the initial and final synchronization point, both hardware and compiler are free to make many optimizations, as long as they don't introduce the spurious writes of Example 2. Writes to shared variables can be done out of order. Writes can be postponed and redundant writes can even be removed. At the hardware level, caches may postpone write-back to main memory. All that matters is that the main memory is made consistent when the synchronization happens.

OpenMP has a memory consistency strategy that is similar to the micro-threaded model (no unsynchronized communication through shared memory), but additionally sports the concept of *flushes*. A flush is a synchronization point between two or more processes. This implies that a processes must wait at the flush until all other processes have also arrived there. Additionally, processes cannot keep any information about shared variables across a flush. Processes are required to write any cached information about global variables back to main memory and no information (like the actual value, but for example also knowledge about a variable's value-range) can be kept across a flush. So on a use after the flush, the variable must be read from main memory. Together this implies that all processes involved in the flush agree on exactly the same state for all global variables at that point. As a result, the value of a global can be passed from one thread to another

by using a flush. Here is the example of such a safe communication:

```
shared variable a ;
PROCESS 1: { a = 25 ;
             flush() ;
           }
PROCESS 2: { flush();
             t = a ; /* t is now 25 */
           }
```

Not discussed here is how it is decided which processes are involved in a particular flush operation. It is assumed there are mechanisms to do this.

For the compiler, flush is relevant because optimizations, in particular related to caching of shared variables, are possible between flush points. No information about global variables is allowed to pass beyond flushes.

6.4 Memory Consistency Models for CoSy

Three memory consistency models are considered for CoSy. These are *zero-consistency*, *sequential consistency* and *relaxed consistency*.

6.4.1 Zero Consistency

Zero consistency is the standard model of CoSy, which is geared toward optimization of sequential, non-parallel, code. In this model all optimizations are allowed that are correct for sequential code. This includes the transformations of Examples 1 and 2.

6.4.2 Sequential Consistency

Sequential consistency is a well known theoretical consistency model. It requires that for every possible parallel execution order their exists a linear interleaved order of all (original) actions in the program. In this order the values of shared variables are well-defined and equal for every process in the parallel execution.

The requirements of sequential consistency are very strong and limit the compiler in its optimizations. For example, two assignments to two different shared variables cannot be re-ordered because they have to show up in the interleaving in the same order as was specified in the source program. The attractiveness of a sequential consistency model is that algorithms that are based on it can be written in such a way that they require far less synchronization than algorithms that do not have that guarantee. Potentially, this reduces the overhead of such synchronization and makes them run faster. This benefit, however, has to be offset against the loss of efficiency through reduced optimization. The jury is still out on which of these has the upper hand. It is known that writing correct programs that avoid synchronization based on sequential consistency requires strong reasoning about the possible states of the program and is potentially error-prone, on top of the usual difficulty of writing correct parallel programs with explicit synchronization.

Although sequential consistency is not relevant for the micro-threaded architecture and also not currently commercially relevant, it is easy to achieve in CoSy by simply making all shared variables "volatile". Volatile has the same meaning in C as in CCMIR. It means that all reads and writes that appear in the (sequential) source program must happen in exactly that same order (and amount) in the executing program. Also, the order (traces) of all volatile variable accesses merged together remains exactly the same after compilation as in the (sequential) source program. This is exactly what is required for sequential consistency of parallel processes.

It is not required to change the source program to achieve this effect. A simple CoSy engines can make all shared variables volatile behind the scenes.

6.4.3 Relaxed Consistency

Th compiler constraints for relaxed consistency are not so hard to define, but they may not be trivial to implement. Relaxed consistency is the model that is required to support the micro-threaded architecture.

To support relaxed consistency for parallel programming, a compiler optimization for sequential program fragments may not:

1. change the sequential semantics of the fragment;
2. move read or write operations of shared variables across synchronization points (like flush, thread creation and termination);
3. keep information about shared variables across synchronization points;
4. introduce a write to a shared variable between two synchronization points in the dynamic execution of the program if there would not have been a write without the optimization.

In these constraints "variable" should be interpreted to mean "memory location". The difference is important for aggregate data, like arrays, where the constraints apply to the single elements of the aggregate, rather than the aggregate as a whole.

The first constraint is a catch-all to make sure that the optimization was correct in the first place.

The second and third constraints are relatively easy to handle by treating a synchronization point similar to a call to an unknown function. Unknown function calls must be assumed to use and modify global variables, so a sequentially correct transformation cannot move read and write operations, or keep information, across the call. Therefore shared variables can be treated the same as global variables around function calls.

The fourth constraint handles the cases of Examples 1 and 2. Both examples introduce a write that was not in the source program.

The fourth constraint is the most difficult to implement since there are many valid optimizations for sequential programs that move statements around. A compiler for a parallel architecture that would simply skip all such optimizations would miss a lot of optimization opportunities.

Regarding the optimization of Example 1, optimizations must be checked for specific optimizations of loads and stores. These are quite specific and should not be too hard to find.

Regarding the optimization of Example 2, this is a more general case of moving instructions in the control flow graph. A possible formulation of "allowed transformations" is:

- ◇ A write statement may move up in the control flow graph if the new location is post-dominated by the original location.
- ◇ A write statement may move down in the control flow graph if the new location is predominated by the original location

These state that if the write is executed on the new location, the old location is also on the execution path, therefore these transformations are valid under the fourth constraint.

Cases exist that are valid under the fourth constraint but that do not fall under the allowed transformations. An example is a write that exists on both sides of an if-the-else statement. Hence we are still looking for a more general formulation of allowed transformations.

7 Summary

This deliverable presents the necessary modifications which are to be done in *CoSy* in order to facilitate the development of $C2\mu TC$ compiler. Most of the engines can be used without any modifications or extensions to be necessary. However, due to the special requirements of the project

some modifications–extensions are necessary mainly to the front–end and to the *Pirctoc* engine while the dependence loop analysis engine needs quite a lot modification or development from scratch. In addition, the CoSy's *Process Model* is discussed, an extension of CoSy inspired from appleCore project, which might be very useful for future extension of $C2\mu TC$.