S(o)OS

Service-oriented Operating Systems

First Implementation Set: Execution Management

Project Deliverable D4.2

Vincent Gramoli [EPFL], Tommaso Cucinotta [SSSA], Lutz Schubert [HLRS], Daniel Rubio Bonilla [HLRS], Aleksandar Dragojević [EPFL], Joao Paulo Barraca [IT], Bruno Santos [IT], Jan Kuper [UT].

Due date: 30/04/2011
Delivery date: 30/04/2011

SEVENTH FRAMEWORK PROGRAMME

This work is partially funded by the European Commission under
FP7-ICT-2009.8.1, GA no. 248465

Version History

Version	Date	Change	Author
0.1	21/02/11	First TOC	Vincent Gramoli
0.2	24/02/11	Initial content	All
0.3	07/03/11	Updated TOC, added EPFL content	Aleksandar Dragojević
0.4	21/03/11	Initial input on real time extensions	Tommaso Cucinotta
0.4.1	21/03/11	Initial input on SAM	Daniel Rubio Bonilla
0.5	23/03/11	Integrated RT-STM and loop parallelization	Aleksandar Dragojević, Daniel Rubio Bonilla
0.6	29/03/11	Updated input on real time extensions	Tommaso Cucinotta
0.7	30/03/11	Integrated Code Analysis and Segmentation	Lutz Schubert
0.8	08/04/11	Updated input on SAM	Daniel Rubio Bonilla
0.9	11/04/11	Input on Programming Models	Jan Kuper
0.10	13/04/11	Input on Code Migration	Bruno Santos
0.11	18/04/11	Updated Code Analysis & Segmentation	Lutz Schubert
0.12	26/04/11	Fixes and minor updates	All
1.0	30/04/11	Fixed minor issues and formatting	Aleksandar Dragojević

Table of Contents

List of Figures

List of Tables

ABBREVIATIONS

Abbreviation	Description
S(o)OS	Service-oriented Operating System(s)
SSI	Single-System Image
TM	Transactional Memory
STM	Software Transactional Memory
QoS	Quality of Service
OS	Operating System

I. Introduction

Management of execution addresses all aspects related to handling code and data in the distributed environment – this includes all issues from segmenting the code over distributing the segments to their execution. Accordingly, this work also covers aspects related to extending the programming language with segmentation related annotations, such as concurrency and dependency identifications.

1. Organization of this document

First, we describe the programming models that ease and improve parallelization of the code (Section II). To this end, we introduce the programming models and describe how they can be used to improve the usability of parallelization techniques. Next, we proceed to describe code analysis and segmentation which is important for increasing code scalability and heterogeneity, and is also very relevant for the usability of proposed techniques. We follow with the higher level, mathematically-based and thus more abstract view of the parallelization programming models.

The next section discusses the most appropriate programming language extensions that enable easier and, possibly, higher performance use of transactional memory and real-time programming abstractions. We also describe the programming language extensions that specify concurrency to increase the resource usage and overall system utility and performance (Section III).

We also describe the speculative execution in detail, focusing on transactional memory, real-time extensions for transactional memory and speculative memory accesses, which is also known as thread-level speculation (Section IV).

Finally, we describe the operating system kernel primitives for code and resource migration across different CPU cores, possibly residing on different hosts (Section V).

2. Motivation

We focus on the programming models at this stage of the project because having appropriate programming models has a potential of greatly improving usability from the perspective of users of future distributed, many-core systems. Choosing the right programming models also promises to improve scalability and heterogeneity of the code as well as help us answer the question of how to use the large-scale systems in different cases.

Devising suitable programming language extensions can help users of the languages (software developers) use the new techniques in a way that is nicely integrated with the target languages. Furthermore, good programming language extensions can provide information to the compiler and the runtime, which can then use this information to generate the appropriate code, reducing the amount of effort required from the software developers. The information that the developers communicate to the compiler and runtime using the programming language extensions can also be used to optimize the generated code, thus reducing the overheads of the applied techniques and improving the performance of the applications.

Speculation is a very promising approach that can be used to parallelize applications in cases when it cannot be determined statically, upfront (e.g. at analysis or compile time) which parts of the application can be executed in parallel. In these cases, the candidates for parallel execution can be speculatively executed in parallel, where the existing dependencies between the parts of the application executed in parallel are determined at runtime. Where such dependencies exist, some parts of the application are rolled back and executed again when the dependencies are resolved.

Speculation is a very interesting technique as it allows transparent parallelization of various applications. It has the potential of alleviating many problems in parallelization. For example, it allows us to solve the typical problems when using locks (e.g. deadlocks and lock convoying) by replacing the lock use with transactional constructs.

We focus on the real-time computing at this stage of the project as we believe that it is important to think about such timing constraints from the start and avoid the trap of trying to enable real-time programming at the later stage of the project as an afterthought. In addition, it is very interesting to understand what is the relation between real-time constraints and speculative execution and to what extent it is possible to integrate them.

Finally, we focus on code and resource migration as these are very important for transparent operation of the systems that span multiple CPUs distributed across different physical machines, that are potentially spread across large geographical areas. In these conditions, it is extremely important to be able to migrate resources. For example, this enables moving them closer to where they are being used for performance reasons, as well as spreading them across large areas for reliability and availability reasons.

We do acknowledge that many other interesting topics related to distributed execution management exist and that the techniques we focused on so far are not necessarily the best fit for all existing workloads. We plan to address some of the other paradigms and techniques in the later stages of the project and when we can find the appropriate for them in our work.

II. Programming For Concurrency

There are two major approaches towards parallel programming notable:

1. Automating all parallelisation, leaving the user essentially with a completely unchanged programming language, and
2. Leaving all parallelisation and optimisation effort to the developer, by providing him with a tool to simplify specific recurring commands , functions and tasks (such as related to messaging)

This distinction reflects the two major developer groups, namely the general purpose developer on the one hand, who aim particularly at usefulness and portability, rather than efficiency of execution. And on the other side scientists, researchers and developers of highly efficient code, typically aiming at execution in high performance computing environments.

Processing power obviously also affects general purpose development, as modern applications offer more and more capabilities and can no longer rely on next generation computing systems to compensate for these requirements. Hence, as both developer groups have to investigate into parallel development, it is surprising that so few attempts have been undertaken to merge these two domains in a common environment.

The main reason for this negligence lies in the complexity involved in developing efficient programs, which generally requires additional knowledge about the hardware structure, as well as its specific capabilities and restrictions. These have then to be mapped to the specific features of the program logic. These features however are generally known only to the developer himself and can hardly be detected using automatic analysis methods. Therefore all the effort remains effectively still with the developer, thus contradicting the usability requirements towards the language.

Ideally therefore, the necessary information is conveyed on a high, abstract level without requiring the user to rethink the application logic, respectively to apply hardware specific criteria to the program. The form of extracting such information should hence be simple, intuitive and straight-forward.

The few existing approaches along this line however are not easy to be used, nor, what is more, are they fail-safe.: if used wrongly, the program can fail and even lead to unpredictable behaviour, crashing the whole system - even if the program logic itself is correct. Effectively, these methods are directly derived from specific low-level approaches in the high performance computing domain, thus implicitly bearing its complexity and their requirements towards the developer in terms of correctness and knowledge about the application logic.

Other approaches relieve the low level dependencies and instead try a complete different way of specifying the behaviour in the first instance. This includes for example mathematical descriptions of the program, so as to identify concurrency and dependencies efficiently. On basis of this information, additional methods for parallelisation and loop unrolling can be applied to improve resource usage and thus theoretically performance. However, not only do most of these models imply complete rethinking of the way of programming, also their level of abstraction implicitly removes the relationship to the hardware too much. This means that the overhead for converting the one representation (abstract code) into the other (executable code) may introduce too much additional overhead to cover for this conceptual distance - for example by introducing an interpreter or simulator that effectively takes the commands consecutively and converts them into executable code, thus neglecting many of the additional optimisation techniques that can be implemented to increase efficiency further. Even though the language may be well adjusted to the platform, the efficiency degradation introduced by the overhead is still way beyond the efficiency of low level approaches and only a little better than classical programming languages. Accordingly, high languages such as Java can never reach the degree of performance of C, even though they are fairly easy to use and thus popular.

Purely functional languages face similar issues, but introduce additional capabilities for steering parallelism that makes them interesting for further investigation. As such for example the replacement of the "for" statement with "for each" commands allows the compiler to increase efficiency through out-of-order execution. What is more, the command implies that the operations are effectively independent of each other and thus can be executed in parallel. Microsoft has already shown with the introduction of C# that aspects of declarative and imperative programming can be successfully combined to increase efficiency, and even usability of the programming language.. It must be noted in this context though that C# was not developed with the particular aim of increasing parallelism, but rather usability and manageability of code.

The main point behind these simple paradigm changes consists in introducing information about flow and dependency of data across the execution time. And as these extensions even adhere better to the human way of thinking, classical paradigms may even be considered usage inhibitors in comparison.

Rather than taking an isolated approach, i.e. maintaining the distinction between general purpose and high performance computing developers, it Is necessary to take an essentially holistic view on the problem and hence an integrative view on the two perspectives. This means in particular, that the programming model must incorporate language patterns that are easily and intuitively understandable, yet at the same time enable the compiler to exploit these added features for better and move efficient execution. The two previously isolated views must therefore respect and integrate each other's considerations to build such an integrative model. There obviously exist two principle approaches, following the existing principles of the current language models. That is on the one hand the bottom-up approaches of law level, hardware near extensions, and the top-down considerations of higher languages. Thereby the bottom-up method must assess its model for potential abstraction and in particular for potential means to steer the behaviour on the lower level. Similarly, the top down considerations must test their constructs against their impact and hence efficiency on the lower level.

It is a particular goal of the S(o)OS project to bring these two approaches together to exploit additional information about concurrency and data dependencies, as well as to exploit the specific behaviour features typical for each level.

In the following chapters we examine the type of features, how they can be assessed and aligned.

1. Code Analysis and Segmentation

Concurrency in a system is represented on different levels in different forms. In particular the logical description of the underlying algorithm provides relevant insight into the *intended* concurrency of a system. This implies in particular aspects such as dependence and independence of functions and objects. Even though the intention is thereby typically implicitly rather than explicitly specified as part of the code behaviour and therefore more difficult to extract than in any other form, the main problem rests somewhere else: the implicit dependency information has only secondary impact on the low level execution where instruction level parallelism, cache misses etc. form the major efficiency criteria. Currently, concurrency information provided on the higher code level is also generally lost during the conversion (compilation) process.

As noted in the preceding section, we therefore face two major challenges: (1) maintenance and exploitation of the higher level concurrency information on the lowest level through compilation and (2) vice versa extending the higher level means of specifying concurrency with information that is meaningful on the low level of execution. In other words, we need to align the two levels of description with respect to their intention and usage. This section thereby looks in particular on how concurrency can be exploited on the low execution level in a form that actually increases performance.

A) The Impact of Code Conversion on Concurrency

Almost all high-level programming languages grant the developer to make virtually unbound usage of memory, and not only in terms of data storage, but more importantly in terms of object instances, number and scope of variables etc. Obviously, the developer will want to restrict that usage in order to prevent serious memory problems, such as memory leakage that may crash the whole system. However, the degree of memory restriction and in particular the impact of wrong usage is generally not noticeable, let alone controllable by the user.

In addition to this, information and control events can seemingly be passed effortlessly between processes and devices. On this high level, the impact of latency and bandwidth restriction seems to be a usage specific constant which can hardly be influenced. Protocols and device specific behaviour are transparent and the impact of *timing* synchronisation on the overall execution performance is not only generally hidden, but also difficult to control, in particular if the underlying details are unknown.

Through usage of encapsulation, function invocations and dedicated data objects, the degree of concurrency seems to be implicitly obvious, if carefully programmed. The amount and type of data passed between functions thereby defines the relationship between the according instances and implicitly their degree of dependency, respectively concurrency. For example, let’s look at a program that primarily analyses and visualises a 2dimensional set of data. The program logic will most likely consist of two major logical sections: one for the analysis and one for the visualisation; both sections will iterate over the 2 dimensions of the array and call an analysis, respectively visualisation function on each datum. On source code level, it can be easily noted that the two major loops are unrelated, if they share no variables – once they share variables (here the array, and potentially the x and y coordinates), the dependencies become less obvious and require additional analysis; what is worse, they may share an additional variable n which may introduce dependency, if it is altered by the one block and consumed by the other.

```
for (int x=0; x<100000000; x++)
  for (int y=0; y<100000000; y++)
    g( a[x][y], n );
for (int x=0; x<100000000; x++)
  for (int y=0; y<100000000; y++)
    h( a[x][y], n );
```

Once converted into executable machine code, this level of dependency will become less apparent. The major distinguishing factor thereby is due to the restrictions on machine code level, which include in particular: (1) limitation of registers, (2) restricted command set and (3) restriction of (fast) memory. These limitations lead to serious reorganisation of the code by the compiler in order to match the seemingly unbound use of variables with the restricted amount of registers on the machine code level by reserving an according range of memory and distributing according pointers across the code. In fact this step already requires that at least re-usage of variables have been resolved, i.e. that some concurrency identification steps have been taken on *source* code level.

Obviously, many commands on source code level have to be converted into long operation chains to fulfil the same capabilities, as the complexity between these two levels does not match. In combination with the limited registers, this can lead to multiple memory offloads, stack accesses etc. just for a single high-level operation, if it deals with multiple variables. This constant variable replacement and memory access makes it difficult to identify concurrency in the first instance.

```
     MOV ECX, 100000000
LX:  PUSH ECX
     MOV ECX, 100000000
LY:  PUSH ECX
     CALL G
     POP ECX
     DEC ECX
     JNZ LY
     POP ECX
     DEC ECX
     JNZ LX
... repeat for h ...
```

What is more important, the constant access to memory introduces an additional delay on the execution side. Depending on the speed of memory, this can sum up to a serious impact, where execution is slowed down by a constant factor. Ideally, the process just uses the lowest-level cache of the processor, as this is the fastest possible memory for execution – however, the size of it is restricted to around 512kB, which is way lower than the typical memory requirements of modern day programs. This gets even worse when one respects the actual degree of dependencies with other applications and in particular the operating system, which again will also take away part of the memory.

If the processor has to execute multiple programs at the same time (multi-task-system), the memory will also be constantly flushed by the context switches between the individual tasks which increase the delay even further.

In the worst case, i.e. where the program always tries to access memory beyond the cache scope, every access will lead to a cache miss, which implicitly increases program execution time significantly. In order to reduce the impact of memory access, most compilers already try to reorganise the data structure cleverly so as to make sure that frequently accessed data is blocked together. However, much of the information that could be exploited to this end is just not known by the compiler, such as usage restrictions of a variable (in particular if it's effectively a constant), usage scope (when can the variable be unloaded), number of jumps, invocation frequencies etc.

A SPECIAL NOTE ON LOOP PARALLELISATION

A specific aspect that current compilers focus on for the purpose of increasing the level of concurrency – and in this case, the level of parallelism – consists in so-called "loop unrollment": a loop that iterates over multiple values in an array, generally executing the same operations on each field, are ideal for execution on a vectorial or SIMD (single instruction, multiple data) machine. These types of processors effectively execute the same operations on multiple data sources at the same time. In other words, they execute the actual operations of the individual loop iterations in parallel, respectively multiple iterations in parallel. It should be noted thereby that the width of the SIMD (i.e. the number of processing units) defines the number of concurrent executions that can take place.

Obviously, such concurrent execution requires that the iterations are independent of each other – or expressed differently: that the operations can be executed on any sequence of array fields, i.e. the loop definition does not imply the sequence of execution. This means implicitly, that no iteration tries to access values / fields that have been modified in a previous iteration.

This however poses exactly the main problem for the compiler: the degree of dependencies between iterations cannot be easily identified. In the following, we will elaborate the different cases of dependencies that might arise in more detail:

Single Loops – Constant Index Distance

Case 1 - No dependencies

S:	A(i) = A(i) + C(i)
T:	B(i) = B(i) - C(i)

Table 1: Example 1

There are no data dependencies between the different statements inside the loop, this allows a wide range of possibilities for parallelisation:

1. The first option is to split the loop into the different independent statements. This might be easy to implement, but the maximum level of parallelisation would directly depend on the amount of data independent statements, and not on the resources available on the system. It might also happen, as in *Example 1*, that even if there are no data dependencies, they both use common data (matrix *C*). This might imply that the same data must be transferred to different computing units instead of having each one working on a different data set.
2. The second option is to make a distribution of the loop among the different resources and let each computing unit work in a different range of the iterator of the loop.
   This way is more flexible to adapt the level or parallelisation to the amount of resources available, and also to the performance of each computing unit.
3. Also a combination of techniques *a)* and *b)* can be used simultaneously.

Case 2 - True dependency (same iteration)

S:	A(i) = A(i) + C(i)
T:	B(i) = B(i) - C(i) + A(i)

Table 2: Example 2

In this subsection the case where a true data dependency between the statements of the loop in the same iteration is analysed. As shown in *Example 2*, the execution of statement *T* depends on the result of *S*. The ways of parallelising this kind of loops are:

1. Split the loop in the statements *S* and *T*. First complete the execution of *S* along all the iterations and then execute *T* so that it can make use of the results of *S*. The parallelisation will be obtained from the individual loop distribution of the loops *S* and *T*. The individual characteristics of the statements *S* and *T* should be considered to achieve the best performance according to the resources available at each moment.
2. The second option is to make a distribution of the loop among the different resources and let each computing unit work in a different range of the iterator of the loop.
3. Also a combination of techniques *a)* and *b)* can be used simultaneously, but this would imply that some synchronisation method is needed. If S_i is the work package distribution for the statement *S* in the range of the iterator of the "chunk" *i*, then S_i and T_i must be synchronised so that T_i starts only when S_i is finished.

Case 3 - True dependency (with previous iteration)

S:	A(i) = A(i) + C(i)
T:	B(i) = B(i) - C(i) + A(i-d)

Table 3: Example 3

Another case is when there is a true dependency but with a data calculated in a previous iteration of the loop. In *Example 3*, the statement *T* depends on the execution of the statement *S*, of d iterations before (*d* must be greater than 0).

The different ways to obtain parallelisation of this kind of dependencies are:

1. As in the first option in *Case 2*, the simplest solution is to split the loop into two, one with statement *S*, and the other with *T*, and run one after the other. The parallelisation would be obtained by the loop distribution of each previously split loop.
2. But in heterogeneous systems it might be more interesting to run each statement in different computing units, and that they all run simultaneously. The whole range of the iterator is divided into different "chunks". S_i and T_i must be synchronised so that T_i starts only when S_i is finished and the last part of S_{i-1} has been calculated (to access to the data of previous iterations spaced by *d)*. The ways of doing this synchronisation are:
   a) Start the execution of T_i only if S_i and S_{i-1} are completely finished.
   b) Si calculates the *d* last elements of S_{i-1}, and T_i starts executing only after S_i finishes executing. This option might simplify the synchronisation communication.
   c) S_{i-1} starts with the execution of the last *d* elements of it range, and T_i starts executing only when S_{i-1} finished the last elements and S_i is completely finished.

Case 4 - Antidependence (WAR)

S:	A(i) = A(i) + C(i) - B(i)
T:	B(i) = B(i) + C(i)

Table 4: Example 4

To solve antidependencies (Write-After-Read) inside the same loop iteration a few options are proposed:

1. As there are no data dependencies between different iterations of the same loop, the loop can be distributed in "chunks" between the difference resources.
2. Another approach is to split the loop into different loops for statements *S* and *T*. This can be achieved with renaming. In *Example 4*, it is necessary to modify statement *T*, so it does not write in matrix *B*, but in B_2, but after execution *B* must be swapped with B_2.
3. Also alternatives *a)* and *b)* can be combined to achieve loop splitting and distribution.

Case 5 - Antidependence (WAR) (with increased index)

S:	A(i) = A(i) + C(i)
T:	B(i) = B(i) - C(i) + A(i+d)

Table 5: Example 5

The last single loop case analysed is antidependencies (Write-After-Read) in the next *d* iterations of the loop (in *Example 5 d* must be greater than 0). The proposed alternatives are:

1. The loop can be split into different into individual loops for reach statement, by using renaming, as in *Case 4b*.
2. The use of renaming and then loop distribution among computation units similar to *Case 4c*.

Single Loops – Varying index distance

Case 6: Antidependence (WAR)

```
S:    A(i) = A(i) + C(i)
T:    B(i) = B(i) - C(i) + A(i*f)
```

Table 6: Example 6

The first single loop with varying index distance analysed is an antidependency (Write-After-Read). In it, the index for accessing array *A* in statement *T* increases faster than in *S* (in *Example 6 f* must be greater than 1) and *i* should be greater than 0. In Figure *1* how the two different indexes increase along each iteration of the loop is shown. It can be observed that after some iterations the index of statement *S* (*ind1*) will never reach the values of the index of statement *T* (*ind2*). The are many alternatives alternatives to solve this dependency, some of them are:

1. The loop can be split into different into individual loops for reach statement, but the loop of statement *T* has to be executed before the loop of the statement *S*, at least till the index in *T* is greater than any value that the index in statement *S* will obtain.
2. As the approach in *Case 3* we will divide the iteration range in chunks. The execution will be synchronized so that *Si* is executed only after T_i and T_{i-1} finish running. (After reaching the point where the index in *T* will always be greater than any possible value for the index in statement *S*, we no longer need this synchronization.)
3. As this is a Write-After-Read dependency, we can solve it also renaming the name of the array *A* in which the statement *S* is writing.

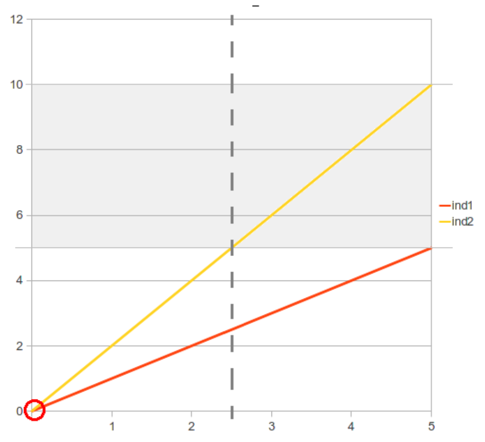


Figure 1: Case 6, indexes (ind1:S, ind2:T, f=2)

Case 7: True dependency

```
S:    A(i+s) = A(i+s) + C(i)
T:    B(i) = B(i) - C(i) + A(i*f)
```

Table 7: Example 7

This case shows a True dependency among the statements *S* and *T*. In the *Example 7* (where *s* is greater than 0, and *f* greater than 1), the distance between the indexes for accessing array *A* in statement *S* and *T* is decreasing till at some point, they will have the same value. In Figure *2* is marked in gray the range of value of indexes for *T* which the indexes of *S* will never reach. Note that in the case in which both indexes have the same value is solved here and not in *Case 6*. The proposed alternatives for solving this kind of dependencies are:

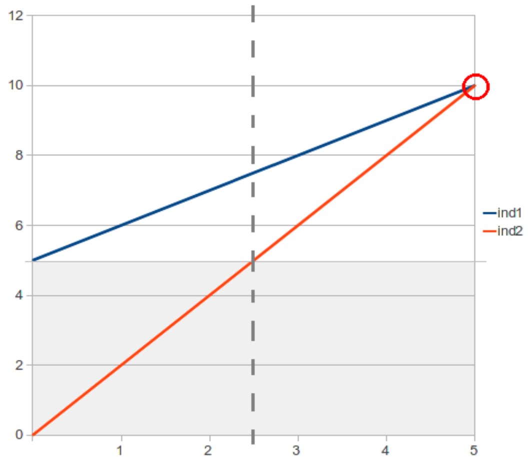


Figure 2: Case 7, indexes (ind1:S, ind2:T, f=2, s=5)

1. The loop can be split into different into individual loops for reach statement, but *S* have to be executed before *T*, except for the range of indexes of *T* that the indexes of *S* will never have, that we can execute before, as their input data is not affected.
2. This case is illustrated by Figure *3*. It can start with running the statement *T* in the indexes that are not affected by the result of *S* (gray area). At the same time the range of indexes of *S* is divided into chunks. As there are no data dependencies in *S*, we can parallelise each chunk. After a chunk of *S* finishes being processed, then the statement *T* can be executed for that range of indexes for accessing the data in the shared array. The execution of *T* here can also be parallelised. It has to be noted that when it is written about range of indexes, we are not referring to the iterated index *i*, but we mean the real indexes that access the shared array, in this example the values of *i+s* for statement *S* and *i*f* for statement *T*.

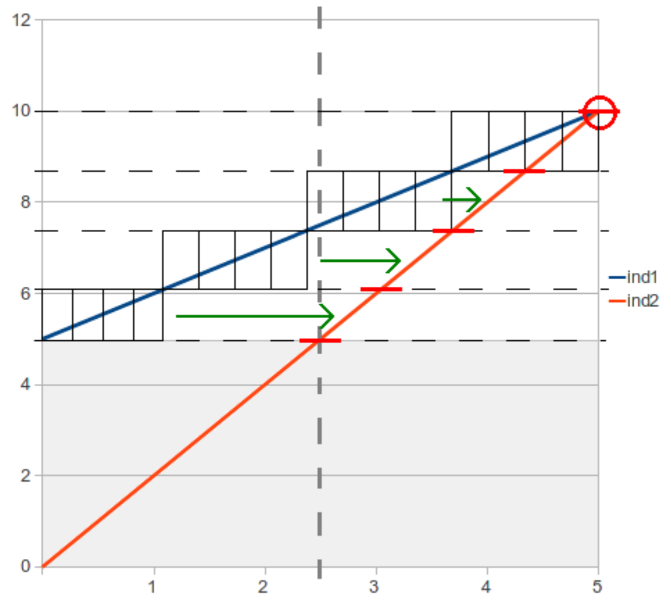


Figure 3: Case 7b, indexes (ind1:S, ind2:T, f=2, s=5)

Case 8: True dependency + Antidependence (WAR)

```
S:    A(i+s) = A(i+s) + C(i)
T:    B(i) = B(i) - C(i) + A(i*f)
```

Table 8: Example 8

This case is an extended version of *Case 7* in which the range of indexes make an inference point in the order of how the statements *S* and *T* access the common data array. The proposed way to solve this kind of dependencies is to detect the inference point and execute the range of indexes of before and after the inference point as different cases. As seen in Figure *4* (*ind1 = S* and *ind2 = T*) two zones (*A* and *B*) can be defined. Zone *A* can be resolved as a *Case 7*, and zone *B* can be resolved as a *Case 6*. The inference point must be included into the execution of the zone *A*.

Case 9: Antidependence (WAR) + True dependency

```
S:    A(i*f) = A(i*f) + C(i)
T:    B(i) = B(i) - C(i) + A(i+s)
```

Table 9: Example 9

This situation is very similar to the previous *Case 8*, being an extended version of *Case 6* (the range of accessing indexes make an inference point in the order of how the statements *S* and *T* access the common array of data). The main difference with *Case 8* is that the way the two statements access the array is inverted. This causes, as show in Figure 4 (*ind1 = T* and *ind2 = S*), that zone *A* corresponds to a *Case 6*, and zone *B* to a *Case 7*. The inference point has to be calculated in the execution zone *B*. The proposed solution is to resolve zone *A* and *B* individually.

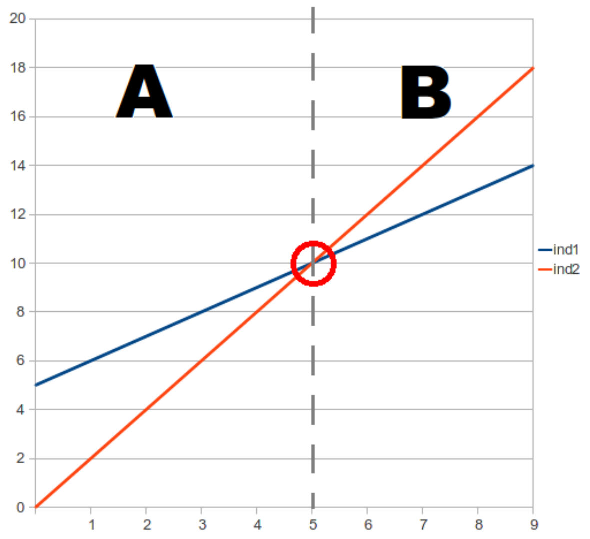


Figure 4: Cases 8 & 9 (f=2, s=5)

B) Analysing Machine Code Behaviour - Principles

One of the essential questions is therefore, how much information can be extracted from the machine code that can still be exploited for the purpose of increasing efficiency, and how much of this information can be reused on the source code side to increase compilation performance in the future.

Given the statements in the preceding section it becomes obvious that performance is significantly influenced by the structure and organisation of the machine code, in ways that are not visible from the source code level. Even though the compiler tries to compensate most of these issues, it still suffers from essential information that is missing from both levels. From the machine code perspective, in particular additional information about usage behaviour would help improve the compiler steps. What is more, compilers currently do not aim for isolation and creation of concurrency, but for best (sequential) execution speed – the impact of cross dependencies is therefore of little concern for them.

The primary goal of machine code analysis therefore consists in identifying these dependencies and in trying to exploit them for increased parallelism. This should not be confused with automatic parallelisation, even though it aims at similar goals (namely parallel execution): Parallelisation tries to convert the (sequential) implementation of a logical function, which could even be a mathematical kernel, into its parallel representation. In other words, parallelisation looks at altering the code logic – unrolling loops is the typical example for this task. In Amdahl's law, parallelisation tackles the parallelisable portion of a code. As opposed to this, concurrent execution tries to identify unrelated code segments so that they can be executed in parallel. In the simplest example, executing two unrelated programs at the same time is a form of concurrent execution. The interesting thing about concurrency identification lies therein that it tries to improve the performance of the *sequential* code portion in Amdahl's law. It should be noted thereby that parallelisation and concurrency share a large common field, depending on the granularity examined: as such, the iterations in a loop can be regarded as unrelated / independent code sequences, which can be executed in parallel due to their level of concurrency.

In these sections we examine in particular how to identify concurrency on machine code level.

PRINCIPLE METHODOLOGIES

Just like with general code analysis, there are two main principles to analyse code behaviour: (1) online analysis that is executed at runtime of the code, thus providing additional information about the actual amount of loops to be executed etc., but impacting on execution time and being more complicated to generate information across the whole program, and (2) offline analysis, which takes the static code and analyses it for dependencies etc. Offline analysis is typically in cases where generic conversions have to take place, i.e. where the results should be independent of the specific data that is fed into the code. Compilation typically applies offline analysis mechanisms. As opposed to this, online analysis is hardly ever employed, as there exists too little information about how to use, how to interpret the results and how much overhead is produced by the mechanisms to monitor memory and register access etc., even though modern chips more and more seem to offer the base capabilities for such monitoring.

Modern mechanisms rely more on software *profiling* to increase the performance of code than on online analysis. However, profiling typically gathers more high-level type of information which serves more as an indicator for specific problem types, such as communication bottlenecks, concurrent access to harddrive etc. This information can be used to explicitly hunt programming issues, respectively to identify the best hardware setup for its execution (this is related to benchmarking).

Offline Analysis

In the section below, we will elaborate on the specific mechanisms proposed by S(o)OS – here, we will first of all provide a general overview over how offline analysis is executed and what kind of information it provides. Offline analysis generally works on the (text) file of the program code – this could either be the source code or the disassembled machine code, or even any intermediary output in-between, if e.g. the algorithm should make use of the analysis steps already taken by a specific compiler.

The text is then analysed with respect to the different forms of dependencies that exist within a (stateful) workflow, that is:

1. Work Flow Dependencies
   The primary dependency in a program is defined by its actual work flow, i.e. the sequence of operations as they are intended to be executed. In a sequential program this is primarily defined by the order of commands in the source code – however, jumps, calls etc. make the actual workflow more complicated. Accordingly, the analysis has to look for all potential branches in the code, including any type of loop. Due to the nature of static analysis, the analysis provides no information about the frequency or amount of invocations in any direction.
   Data flow dependencies are actually in themselves quite meaningless, if they do not imply other dependencies (data or state) – for example, the two consecutively executed functions g and h in the sample code above do not imply dependency from one another, just because they are executed in sequence.

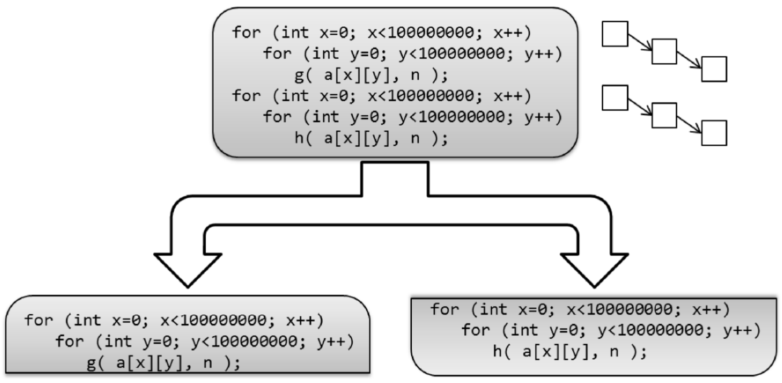
2. Data Flow
   More meaningful than the work flow are the dependencies introduced through shared data or common memory usage. Whilst this is comparatively easy to denote for function invocations on the source code level, this type of dependency is more difficult to identify if the data is passed indirectly, i.e. if it is stored in a common memory space or in a global variable etc. Again, due to the nature of static analysis, the values of the data will not actually be known, so that some dependencies may be misread – this ranges from loop sizes over conditional branches to memory access indices. In order to ensure correctness,

each branch in the workflow model has to be equally pursued during analysis.
As we will see below, almost all data dependencies are hidden in shared memory access on the machine code level. As this implicitly hides scope information from the analyser, this makes data dependency assessment on this level even more complicated.

3. State Dependencies
The most relevant type of dependency however consists in shared state across parts of the code - this is also implicitly the actual type of relationship that makes work flow dependencies relevant for analysis: a code section may share a specific state if a variable is assigned once and re-used multiple times. In so far, state dependency is closely related to scoped data flow dependencies, i.e. where the data dependencies are restricted to a specific region and set of variables.

By creating a graph over all the operations and their dependencies, one can identify the relationship and hence the degree of concurrency between individual segments. By assigning weights to these relationships that reflect the nature of dependency (as listed above), this graph can be used to identify more and less isolated segments and to assess the impact this segmentation would have on execution.

Obviously, the type of dependency has different impact on execution – for example, a pure work flow dependency with no shared data or memory could be split at any point without affecting the execution performance. The main point in this case consists therein, that the two segments could be executed in parallel due to their degree of concurrency, without leading to any performance loss of the individual segment.



```
for (int x=0; x<100000000; x++)
    for (int y=0; y<100000000; y++)
        g( a[x][y], n );
for (int x=0; x<100000000; x++)
    for (int y=0; y<100000000; y++)
        h( a[x][y], n );
```

```
for (int x=0; x<100000000; x++)
    for (int y=0; y<100000000; y++)
        g( a[x][y], n );
```

```
for (int x=0; x<100000000; x++)
    for (int y=0; y<100000000; y++)
        h( a[x][y], n );
```

Figure 5: parallel execution of concurrent code segments

As opposed to that, data or state dependencies imply that the segmentation and distribution of the according segments can only take place, if some mechanisms for communicating the content of the data are in place. This is the classical use case for message exchange across threads, as specified e.g. in MPI. This type of communication automatically introduces an additional delay to the segments, not only due to the additional commands, but in particular due to the messaging latency.

In the distributed case, additional delay is further caused by potential misalignment between the segments / threads, i.e. by the fact that the recipient of a message has to wait for the sender if the latter starts sending later than the first starts with reception.

In general therefore, just by adding the communication mechanisms to the code, execution efficiency becomes seriously impacted. However, if the relationship is fairly distant, the segmentation implies that the two parts could be executed mostly in parallel.

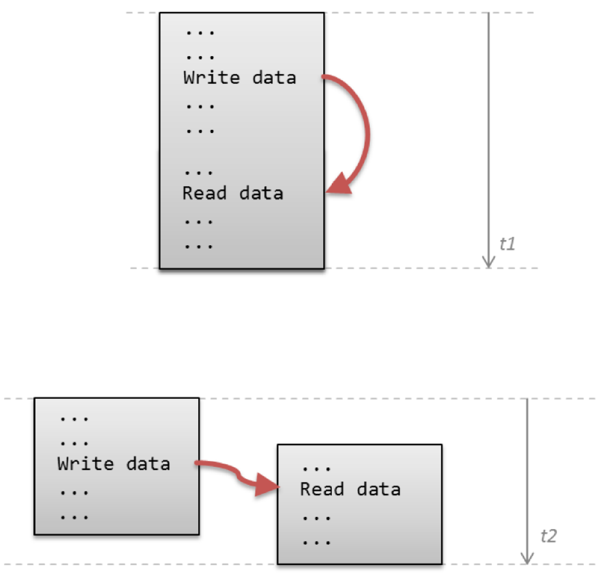


Figure 6: degree of concurrency and the impact on execution timing

In **summary**, the information gained from offline analysis of the code allows identification of relationships which can be used for concurrency exploitation. At the same time however, offline analysis does not provide any information about the actual usage behaviour and is restricted to unloaded data, thereby potentially missing additional information about strength and sequence of dependencies.

Online Analysis

Online analysis pursues the same principles and goals as offline analysis – however, as data is gathered on the fly at run time, the monitor can never be sure whether the relationship graph is complete, before the code execution has not ended. What is more, online analysis should be able to provide some information prior to execution end in order to actually make use of the potential performance gain. As opposed to offline analysis, online analysis also has the advantage of exposing additional information about the code related to its actual execution behaviour.

Obviously, online data can only be gathered on the level of executable code – even though “executable” does not necessary imply machine code level. In the case of Just-in-Time compilers, the source code takes a form of executable that may actually suffice for analysis purposes, though as mentioned above, the information granularity may be too high to take meaningful decisions. Notably, some JiT compilers (e.g. .NET) will maintain a compiled state in memory, so that the executable code is actually close to machine level.

In order to gather online data, the code needs to be executed in a pseudo-virtual environment – depending on the range and type of data interested in, this environment needs to intercept, respectively interpret the according commands and turn them into dependency information. In general, a virtual memory environment will be sufficient to gather the relevant information.

The principle relationships are thereby identical to the ones in the offline analysis, with the addition that timing and repetition information can be used to alter the weights. In particular repetition of invocations (jumps, calls) implies that the weight of the relationship increases accordingly. Looking at an example, where a function body performs m operations over the variable x, but only n over another variable y, i.e. m >> n. In this case, it is obvious that a delay d in accessing x impacts as an overall delay of m*d, whereas the same delay in accessing y leads to n*d increase in execution time. With m >> n it is clear that also m*d >> n*d.

In another case, a process might first call a function g then iterates over the inner loop calling a function h per iteration, clearly shows that the relationship to h is higher than to g. In other words, if g is stalled by delay d, the overall process is delayed by d; if however h stalls by the same delay, the overall process is stalled by the number of iterations (n) over h times d. Obviously, we thereby imply that the loop over h cannot be parallelised, as this would alter the timing relationships again. This is an important additional information which particularly depends on the platform the system is running on, as the number of parallel executions is restricted by the number of available cores (and the degree of parallelisability of the according loop). Accordingly, if only c=2 cores are available, the full execution time is not delayed by n*d, but only by something more than d*n/c. Under the simplified assumption that c cores speed up d by factor c, the speedup becomes neglectable once n reaches c – in a more realistic case, we might also have to distinguish between d for g and d for h. This however belongs to fine tuning the segmentation and will therefore be ignored for the time being, as we concentrate on gathering and interpreting the information first.

In the case of online analysis, all data is gathered on-the-fly and therefore also needs to be evaluated as it becomes available. This means that segmentation decisions are also taken on-the-fly, which however implies that the code is already hosted on a core and would potentially need shifting and rearranging. The according effort should thereby only be undertaken, if a performance improvement is very likely and exceeds the implicit costs for code maintenance. As the measurement must be incomplete at the time of assessment, the cost / gain analysis must implicitly be a pure estimation.

However, there are nonetheless indicators for potential segmentation and distribution that can be gained from the offline analysis interpretation. Basic parameters relate to the space occupied in particular in cache, amount and likelihood of branching given the time frame, and in particular creation of new context space by reinitiating the registries etc. (see also discussion below).

The gathering method thereby follows a reverse principle of the offline analysis, i.e. it executes the cut phases prior to full data gathering.

General Principle

In general, information about the code behaviour is gathered and analysed in a multi-phase process in order to ensure completeness. As noted, this process cannot be adhered to in the online analysis mode, as no full data is available in this case. Nonetheless, the principle even applies in online analysis, as the partial data collected needs to be treated as complete data with a principal stochastic deviation to account for the inaccuracy of the data.

The principle thereby consists in four major processing steps (which partially execute in parallel):

1. Gathering of the data and generation of a dependency graph as detailed below
2. Merging related / too close nodes in order to reduce the graph size
3. Execute cut algorithms on the dependency graph to identify most concurrent points
4. Repeat until either no further concurrent points can be identified anymore or the cutoff criteria (size & efficiency) are fulfilled

In the following chapter we will elaborate this process in detail for the offline analysis of executable code.

C) *Offline Analysis of Executable Code*

Code analysis as proposed here is executed on a graph representation of the code and its behaviour. We thereby generate a graph from the machine code where code blocks denote vertices and relationships are represented by edges. A “relationship” is thereby any form of dependency between the respective code parts, including

- Work flow relationships, i.e. the normal execution order of commands in the code. Implicitly, this follows the potential movement of the instruction pointer through memory. This includes next to (normal) sequential execution, in particular jumps and calls. It must be noted thereby that conditional jumps are branches in the code execution behaviour, where both branches need to be examined. As noted, work flow relationships are meaningless, if they do not imply some other form of dependency (data or register).
- Data relationships are represented through common access to memory addresses. However, common access to memory is in itself meaningless, for example if all code segments just read the memory and do not alter it (in which case the value becomes a constant). More important are hence the relationships between write and read accesses to the memory – an edge should therefore represent the dependency between a write and the (multiple) follow-up reads of the according address. This represents the data dependencies between code parts.
- Register relationships form the strongest dependencies between different code segments, as they imply the fastest access to memory. Similar to the data relationships, the *scope* of register assignments is of particular relevance for the code analysis, as for example a global register assignment (which is hardly ever the case) still allows for concurrent reads.

On basis of this graph, we can assess the relationship between individual code segments basing on the implicit cost that a communication based data exchange (as opposed to memory or register access) would have on execution performance. The segmentation process thereby generally tries to minimise this cost, rather than avert it through concurrency – this analysis is performed in the next step, where the potential segments are examined for the concrete cost and performance benefits that could be gained (respectively lost) through concurrent execution. A particular aspect to pursue thereby consists in increasing the amount of overlap, i.e. the degree of real concurrency between threads – whereas ideally no thread would have to stall. This assessment must respect the fact that communicating the data implies a messaging delay proportional to the latency between the actual processing units that will execute the process(es). Implicitly, full assessment will require that the platform is known and that a mapping can take place. The initial assessment therefore will focus on the general degree of overlap, rather than on the concrete timing constraints.

STEP 1: GENERATING THE CODE GRAPH

In order to generate the dependency graph of the executable code, the individual operations and their “meaning” have to be interpreted. This can be done by reading the disassembled source file and executing the individual commands consecutively. Notably, the graph generation algorithm should not emulate the full behaviour of the processor, as this would exceed the available time by far. Instead, the algorithm will essentially assess each operation for its potential relationships across the code, thus ideally taking a complexity of O(n); where n is the number of lines of code. Implicitly, the relationship definition may introduce errors in the case of indirect memory access, i.e. where the actual destination valued is stored in a memory address, which in turn is accessed by a load directive – as the algorithm does not reflect the actual memory content, the final value cannot be resolved. This however only impacts on the graph in the case of indirect jumps, where the destination is hence unknown – the implicit additional latency for reading a value has, however, only minor impact on the weighting of the edges and thus can be neglected in the graph.

The algorithm is thereby straight-forward. Given a *weighted* graph G = { V, E }, a list R consisting of as many 2-dimensional elements as there are distinct registers times their byte size[1], a function $q{:}r \rightarrow \{l\}$ that maps a register name r to all the elements l in R that belong to that register[2], empty sets M, J, C and a counter t=0. Also, all commands are stored in a list P.

Start from the code entry point.

For each instruction address i add i to V. Analyse each command c for the following conditions:

- If c executes a write on memory, store the instruction address i_{write} and the destination memory d_{write} as a two-dimensional $\{i_{write}; d_{write}\}$ object in M. If M contains an object {i; d} where $d=d_{write}$ and $i \neq i_{write}$ then remove {i; d}.
- If c executes a read from a memory address d_{read}, search in M for {i; d} where $d=d_{read}$. Create an edge $\{w_{data}, i_{read}, i\}$ where i_{read} is the current instruction address, and add to E.
- If c assigns a register r with any value (i.e. direct, indirect, or another register), store {i, t} in all elements q(r) and increase t by 1
- If c uses the content from a register r, select {i,t} from q(r) where max(t). Create an edge $\{w_{register}, i_{read}, i\}$ where i_{read} is the current instruction address, and add to E.
- If c executes an unconditional jump, change the instruction address i to the jump destination address i_{dest}. Add $\{ w_{workflow}, i, i_{dest}\}$ to E.
- If c is an (unconditional) call to a (direct) address, store the {i,t} in J where i is the current instruction address. Increase t by 1. Change the instruction address i to the jump destination address i_{dest}. Add $\{i, i_{dest}\}$ to E.
- If c is a return command, retrieve {i,t} from J where max(t). Remove {i,t} from J and add $\{w_{workflow}, i, i_{cur}\}$ to E, where i_{cur} is the current instruction address. Change the current instruction address to i.
- If |J|=0 then the end of the program is reached, abort the algorithm
- If c is a conditional jump, store {i,R,M,J} in C. Change the instruction address i to the jump destination address i_{dest}. Add $\{w_{workflow}, i, i_{dest}\}$ to E.

Increase the instruction address by the size of the current command, and retrieve the next command c from the list P. Add $\{w_{workflow}, i_{next}, i_{prev}\}$ to E, where i_{prev} is the previous and i_{next} the next instruction address. Repeat analysis until aborted.

Once aborted, check if |C|=0 – if this is the case, the algorithm has ended successfully, otherwise retrieve {i,R,M,J}, increase the new instruction address i by the size of the (jump) command at i and repeat the algorithm.

For all edges, $w_{workflow}$, w_{data} and $w_{register}$ denote the (constant) weight that implies the cost for transferring the according information as a message, rather than in its current form. In other words the weight indicates the relative increase in latency that would occur from splitting the code along the according edge. Even though the weight for advancing the instruction pointer should thus be

1 More concretely, the list should cover the same size as the registers would require serialized in memory

2 This requires some elaboration: looking at the x86 register set, we have among others the registers EAX, AX, AL, AH, which all belong to the same space. In other words, if we map these registers to an address space (cf. Figure), we notice that EAX occupies 4 bytes, the two lower of which are also occupied by AX, which in turn occupies the space of AH and AL. Implicitly, by changing EAX, the registers AX, AH, and AL are also altered.

31 24	23 16	15 8	7 0
EAX			
		AX	
		AH	AL

highest, the overall impact is nonetheless low (as it requires essentially only a remote event to trigger remote execution). We therefore consider $w_{register} > w_{data} > w_{workflow}$ – a rough indicator is 100 > 10 > 1.

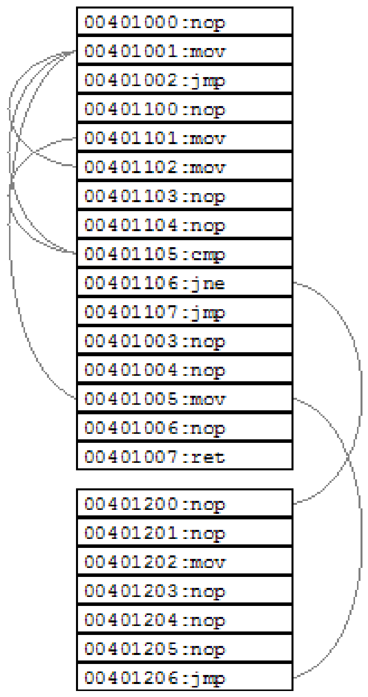


Figure 7: a dependency graph of a highly simplified program. Note that all consecutive nodes are depicted as a full block and the according edges hidden for visualisation purposes

Notably, the given algorithm executes all branches completely, i.e. will iterate through all loops completely, thus also investigating some commands redundantly. By adding a criteria to abort the algorithm once an existing address is revisited, the according redundancy can be avoided – however, this may make the list of dependencies incomplete, for example if a branch overwrites data used in the main function.

STEP 2: REDUCING THE CODE GRAPH

As graphs can get very large with complex programs, it is sensible to reduce the graph size so as to allow for a more effective segmentation and assessment. This step takes some of the segmentation decisions ahead of time, which essentially means that nodes that should not be separated will be merged right away. One approach would consist in executing the first iterations of the segmentation process over different nodes (see below), but the cut-off criteria would be too unclear to not potentially impact on the segmentation quality. Instead, it can be noted from a typical program structure that certain blocks belong and should stay together. The primary criterion in all cases is thereby the scope of a value assignment within the code block: often enough, registers are only assigned to be used within a very limited scope, such as for the purpose of comparing specific values or for moving information into memory etc. Similarly, memory access may just serve the purpose to create an intermediary value place holder etc. Given the cost for messaging (overhead and latency etc.), breaking such small scope dependencies has a too negative impact on execution, i.e. they should not even be attempted in the first instances.

Contrary to intuition, directly consecutively executed operations are not implicitly strongly coupled to one another, even though a good programmer will want to avoid having a mixture of unrelated commands within a code block. Notably, a lacking relationship on the lowest command level, like in two consecutive commands does not imply independence of the commands on a higher hierarchical level – for example, if one command effectively sets the abort flag of a loop, it may not seem related to the calculation that actually forms the workload of the loop, but is nonetheless essentially connected to the total process. The decision to relocate the according tasks therefore primarily depends on the scope of the overarching process, i.e. its workload in relationship to the delay introduced by the relocation (and implicitly, its degree of concurrency).

Detecting Usage Scope

In the simplest case, the scope of a value is directly limited by the usage operations following the assignment – as can be seen, the assigned register is only read by the consequent instructions before it is reassigned again. In the graph this shows as a limited range of edges related to the assignment of the according register. In such a simple case, the scope can even be assessed on the fly, as the register is overwritten before it can be used out of scope.

```
     MOV ECX, 10
     XOR ECX, EAX
     DEC ECX
     JNZ L1
     MOV EAX, FFFE
L1   MOV ECX, 21
```

In the more general case, the graph as generated in step 1, clearly indicates the usage scope of a register as a set of incoming edges from the node that assigns the according register. If the furthest distance does not exceed a certain range, the nodes within the scope can principally be merged.

Notably, even nodes that contain outgoing edges to (read access from) remote nodes may principally be merged, as the actual cut executed below will just indicate the relationship with respect to “nearness” of the segments – it does not imply that the individual cuts should or should not be segmented further. In other words, the original assignment scope would most likely still be maintained as a single block after multiple cut iterations, as the weight within that scope is comparatively high. Accordingly, the merging algorithm could already anticipate this step and maintain the scope as a whole – however, this would implicitly hide the exact assignment location from the later step of evaluating the efficiency of the segmentation (i.e. assessing the degree of concurrency and hence the potential gain in performance, see also Figure 6).

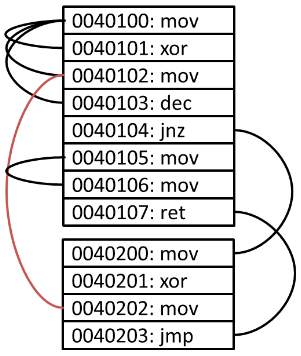
Special Conditions

Two specific cases impact on the node merge based on assignment scopes: (1) branches, (2) multiple assignments / reassignments and (3) overlapping scopes. In either case, the validity scope becomes unclear, as the interaction with other assignments transfers the impact range of the according register beyond the immediate scope.

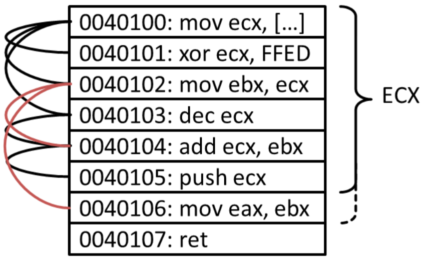
(1) If the code *branches*, both branches may affect the assignment of the registers individually, thus leading to different register dependencies with sections outside the immediate scope. In other words, if the branches assign different values to the same register, it becomes unclear for a later register access where the actual value originates from, if the two branches are merged into a logical block. On the other hand, if a branch accesses a previously set register, this implies again that the respective branch should be close to the register assignment in memory, so that it may merge with the assignment scope. However, this is only advisable, if the accessing branch is local to the scope, i.e. if the accessing branch is not the remote one (“jump taken”) – this is due to the fact that the remote branch typically is less frequently taken than the local branch. Depending on the amount of register accesses from the remote branch, this may however be considered a good indicator to keep

the respective branch local under any circumstance. This decision will be taken by the segmentation step though.

Notably, as the point behind the merge primarily consists in reducing the graph (rather than in creating indicators for the segmentation), it is generally advisable to not merge over conditional branches, unless the code contains a very large number of branches.



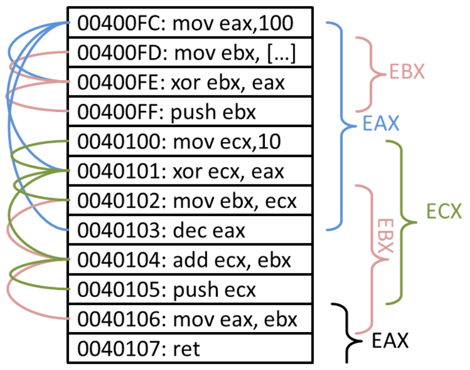
(2) Similarly, multiple assignments, respectively reassignments lead to an unclear relationship between any accessing instance and the assigning operation, just like on higher level. Typically on the machine code level this only occurs when the variable will be used for two different operations, so as to avoid having to read the register source multiple times. From the graph perspective, this means that the register assignment operation contains an incoming edge which points to another assignment, potentially via multiple other nodes of indirection. Effectively, multiple assignments just extend the scope of the respective register, so that node merge can continue beyond the assignment scope of the first register, until any of the other conditions set in. However, from an algorithm stance this means that the register assignments need to be tracked accordingly (respectively, that edges need to be traced beyond the source node)



(3) Multiple assignments are easily confused with overlapping assignment scopes, i.e. where multiple different register assignments come together within a given code block. This is the most frequent case of register usage on machine code level, as the scope of value-to-value operations are comparatively limited in most instruction set architectures. If the individual assignments all maintain a clear scope as in a single assignment, i.e. without branching or remote reassignment (calls) etc. the assignments can be merged into a single block.

The reason for the "mergeability" here is similar to the one in case of a single assignment: as any segmentation between closely linked operations must lead to a performance loss, the overlaps implicitly indicate that no good cut can be found – obviously, this also depends on the degree of overlaps and the distance covered thereby. The figure below shows a case of close assignment overlaps, even though the relative distance of the EAX assignment may serve as an indicator for a potential cut between the first EBX assignment scope and the ECX, second EBX scope. However, in this specific case, segmentation would not be sensible.

Overlapping assignments become particularly complicated, if the scope of any individual register assignment includes a branching operation with concurrent assignments / accesses (cf. case 1 above). In such a case, the merging scope must be treated identically to the case of single assignments containing a branch, i.e. first of all, all other assignments may be merged accordingly, second the affected assignment must be isolated to identify the point of overlap for concurrency assessment.



```
00400FC: mov eax,100
00400FD: mov ebx, [...]
00400FE: xor ebx, eax
00400FF: push ebx
0040100: mov ecx,10
0040101: xor ecx, eax
0040102: mov ebx, ecx
0040103: dec eax
0040104: add ecx, ebx
0040105: push ecx
0040106: mov eax, ebx
0040107: ret
```

Algorithm Principle

The straight-forward approach to merging the nodes in the graph consists in assessing the amount of in- and outgoing edges per node. In general, two nodes n_1 and n_2 can be merged, if n_2 belongs to an assignment scope, in which no branches exist.

$$\begin{aligned}&\forall\{n_0,n_2\}\in E: w(n_0,n_2)=w_{register}\wedge\ldots\\&\exists\{n_0,i_1\},\{i_1,i_2\},\{i_2,\ldots\},\{\ldots,i_k\},\{i_k,n_2\}\in E\,;k<p_{max}\wedge\ldots\\&w(n_0,i_1)=w(i_1,i_2)=w(i_2,\ldots)=w(\ldots,i_k)=w(i_k,n_2)=w_{workflow}\wedge\ldots\\&\neg\exists\, j\in V\,;\{x,j\}\in E\,;x=n_0,i_1,i_2,\ldots,i_k\,;j\neq i_1,i_2,\ldots,i_k,n_2: w(x,j)=w_{workflow}\end{aligned}$$

whereas p_{max} indicates the maximum path from the assigning node n_0 to n_2. If the nodes meet the condition, remove $\{n_1,n_2\}$ from E. For all e in E that are of form $\{x,n_{1/2}\}$ or $\{n_{1/2},x\}$ replace with $\{x,n_{merged}\}$ or $\{n_{merged},x\}$ respectively. Remove n_1, n_2 from V, add n_{merged}. In other words:

- $V'=V\backslash n_1\backslash n_2\cup n_{merged}=\{n_1,n_2\}$
- $E'=E\backslash\{n_1,n_2\}$
- $\forall\{x,n_1\}\in E: E'=E\backslash\{x,n_1\}\cup\{x,n_{merged}\}$
- $\forall\{n_1,x\}\in E: E'=E\backslash\{n_1,x\}\cup\{n_{merged},x\}$
- $\forall\{x,n_2\}\in E: E'=E\backslash\{x,n_2\}\cup\{x,n_{merged}\}$
- $\forall\{n_2,x\}\in E: E'=E\backslash\{n_2,x\}\cup\{n_{merged},x\}$

In light of the previous discussion, this definition seems incomplete, as aspects such as overlapping assignment scope etc. seem to be missing. However, as discussed above, distinguishing the according cases only is of relevance when a branching node leads to potentially false interpretation of the relationship.

More important however, reassignments, respectively multiple assignments are not captured with this formula, as they are easier to tract programmatically. Multiple assignments may extend the scope beyond p_{max} and are therefore principally of additional interest. To this end, every register assignment is stored in a dedicated list whereby the last unique assignment overwrites all previous ones. When resolving an assignment, the according register is verified against this list of assignments, thereby resolving the reassignment. Accordingly, two nodes can merge that have seemingly different assignment scopes.

Repeat the algorithm with G' until G=G', i.e. until no more changes occur.

STEP 3: SEGMENTING THE DEPENDENCY GRAPH

The main step of the code behaviour analysis consists in the actual segmentation in such a way that the cost for introducing communication would be minimal, i.e. so that latency is affected in a least possible fashion. This means that we are looking for a set of subgraphs {H_1, H_2, …, H_n} where

- $H_1 \cup H_2 \cup \ldots \cup H_n = G$ and
- $\forall a, b; a, b < n: H_a \cap H_b = \varnothing$ and
- with $c(i, j) = \sum w(a, b); (a, b) \in E \wedge a \in H_i \wedge b \in H_j \wedge i \neq j$, c(i,j) is minimal for all i,j

Or in other words: the graph G={V,E} is partitioned in subgraphs {H_1, H_2, …, H_n} so that the full cost of all edges across any two subgraphs is minimal. This however is in principle identical to the max-flow min-cut theorem [52] and according algorithms may hence be adopted.

Principles

The typical min-cut algorithm is only applied to a single cut across a graph, so we need to adjust the algorithm to generate multiple cuts until a cut-off criteria is reached. Further to this, in case of the intended usage here, we need to refine the criteria for cutting with respect to the following aspects:

1. size of the segments in relation to the available caches per destination core: ideally the whole segment including data would fit in cache in order to reduce the impact from cache misses and (remote) memory access
2. impact of communication overhead and latency on execution: each segmentation implies additional overhead for communication and implicit latency in acquiring the data – therefore the amount of communication should be kept at a minimum. Even if a cut is considered minimal in light of the complete graph, this does not imply that the cut can be used to increase performance.
3. degree of concurrency: segmentation and distribution of the code is only sensible if some concurrent segments can be identified from it, i.e. if the level of parallelism can be increased.

Whilst the impact of communication (criterion #2) is encoded in the weight of the edges as declared in step 1 above, the size of the segments and the degree of concurrency are not directly accessible in the graph. As noted, the normal min-cut algorithm executes a single cut over G, so that it results in two subgraphs (i.e. n=2). By reapplying the algorithm over multiple iterations, we can further subdivide the resulting graphs so as to meet in particular criterion #1, i.e. until the size of the subgraphs approaches the destination cache sizes. Keeping in mind that at the time of segmentation the actual destination platform may not be known, an average value needs to be applied for this criterion. We can thereby exploit the fact that potentially too small subgraphs may be merged again when beneficial – however, in most cases it will be difficult to reach a subdivision smaller than the cache size that still meets criterion #3.

As for the degree of concurrency, we need to distinguish two potential types of concurrency: (1) total independence and (2) partial overlap:

- Total Independence

In the case of total independence, two subgraphs H_1 and H_2 can be identified for which no two nodes n_1, n_2 with $n_1 \in H_1$ and $n_2 \in H_2$ exist, so that a path from n_1 to n_2 can be build. Thereby we say that a path n_1 to n_2 exists, if

$$\exists i_1, i_2, \ldots, i_k;\ i_1, i_2, \ldots, i_k \in V : (n_1, i_1),\ (i_1, i_2), (i_2, \ldots), (\ldots, i_k), (i_k, n_2) \in E \cdot$$

Note that since G is a directed graph, an edge $\{i_1,i_2\}$ means as much as that the vertex i_2 reads a value that has been modified by i_1. Implicitly, a path from n_1 to n_2 signifies that n_2 accesses a value that in some form has to be modified by n_1 first. In other words, i_1 / n_1 have to be executed prior to i_2 / n_2, so that they cannot be executed in (real) parallel.

Partial Overlap

If however for two nodes n_1, n_2 with $n_1 \in H_1$ and $n_2 \in H_2$ there exists a direct edge $(n_1, n_2) \in E$ with $w(n_1,n_2)>w_{workflow}$ and H_1, respectively H_2 can be split up in 2 subgraphs H_{1a} and H_{1b}, respectively H_{2a} and H_{2b}, so that

- $n_1 \in H_{1a} \wedge n_2 \in H_{2b}$
- $\exists n_{1b} \in H_{1b} : (n_1, n_{1b}) \in E \wedge w(n_1, n_{1b}) = w_{workflow}$
- $\exists n_{2a} \in H_{2a} : \{n_{2a}, n_2\} \in E \wedge w(n_{2a}, n_2) = w_{workflow}$
- $\forall m \in H_{1a}; m \neq n_1 :\ \exists p \in H_{1a}; p \neq m : (m, p) \in E \wedge w(m, p) = w_{workflow}$
- $\forall m \in H_{1b}; m \neq n_{1b} :\ \exists p \in H_{1b}; p \neq m : (p, m) \in E \wedge w(p, m) = w_{workflow}$
- $\forall m \in H_{2a}; m \neq n_{2a} :\ \exists p \in H_{2a}; p \neq m : (m, p) \in E \wedge w(m, p) = w_{workflow}$
- $\forall m \in H_{2b}; m \neq n_2 :\ \exists p \in H_{2b}; p \neq m : (p, m) \in E \wedge w(p, m) = w_{workflow}$

Or in other words, H_1 and H_2 can be segmented so that all nodes prior and after n_1, respectively n_2 are grouped in a separate subgraph (cf. Figure 8).

With this segmentation, we can define the overlap between H_1 and H_2 as o(H1,H2):=min(|H_1|,|H_2|). This means effectively, the amount of steps that can at least be executed in parallel with full resource usage. Most correctly, the delay for communication should be taken into consideration in the form of the total overlap being min(|H_1|,(|H_2|-latency)) – however, as the according information is not necessarily available, we hereby consider first the general case.

Therefore we can define as a third criterion, that o(H1,H2) should be maximal.

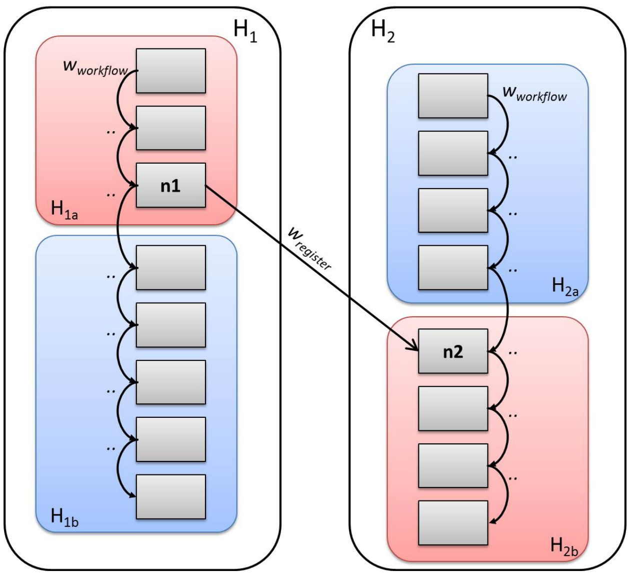


Figure 8: segmentations of two partially overlapped subgraphs

Algorithm Principle

The algorithm used bases on the publication by Stoer and Wagner, which has the advantage of a comparatively short runtime [53]. The principle bases on the same theory as pathfinding mechanisms do: always pursue the path with the minimum total weight, though here we pursue the path with the maximum total weight, so that the remaining edge must the lightest one.

The algorithm therefore executes in an iterative fashion where every iteration reduces the graph size by the nodes eliminated.

Each iteration starts from a subgraph A={a} with a being an arbitrary vertex a ϵ V.

- Select $n \notin A$, so that $w(A,n) = max \{w(A,p) | p \notin A\}$ with $w(A,m) = \sum_{n \in A} w(n,m) + \sum_{n \in A} w(m,n)$; where $w(n,m) = 0 \, if \, \{n,m\} \notin E$.
- Merge n with a ϵ A, so that $\forall x \in V; (x,n) \in E : E \cup ((x,a'))$ and $\forall x \in V; (n,x) \in E : E \cup ((a',x))$.
  Whereby w(x,a')=w(x,n)+w(x,a) and w(a',x)=w(n,x)+w(a,x). A'={a'={a,n}}.
- $E' = E \setminus (\{a,n\} \cup \{n,a\})$
- Replace a, n in V with {a,n}, i.e. $V' = (V \setminus \{a,n\}) \cup \{\{a,n\}\}$
- Repeat until A'=V

In other words: select the node n which is most tightly coupled to (i.e. has the highest total weight with) the elements in A. Merge the two vertices added last and replace them in V and A with the new merged vertex; also merge any edges that go in / out of the last two vertices and remove the edge between the two vertices.

The algorithm ensures that all segmentations that would lead to high communication overhead are ignored from the beginning, i.e. that only weak connections are considered for actual segmentation purposes. In the form above, the algorithm just inspects the weight in the sense of the general communication cost in comparison to the existing method of access – this implicitly leaves aside concurrency as discussed in the preceding section. To encode this into the calculation, we must decrease the actual the weight by the degree of concurrency between the selected nodes.

We therefore either recalculate the full graph prior to executing the mincut algorithm, or reassess the weight whenever the algorithm has to decide between two similar nodes. The latter approach can exploit the fact that the degree of concurrency may actually change with the node merges, as they implicitly specify the parallelism, respectively sequentialism of a code block.

The algorithm will end after multiple iterations with a two-node graph, where each node consists of multiple merged nodes from the original V. The final cut is thereby *not* the optimal one – rather, the optimal cut must be identified by comparing all iterations with each other and selecting the intermediate graph with the lowest partitioning weight. As shown by Stoer and Wagner, the resulting graph is the best bipartite cut for G [53].

Executing Multiple Cuts

As noted, the algorithm results in a bipartite segmentation of G, where both partitions P_1 and P_2 (with $G = P_1 \cup P_2 \ and \ P_1 \cap P_2 = \emptyset$) represent a set of code blocks that would be executed in a single processing unit – therefore the maximum degree of parallelism after a single cut is 2. What is more, the result is not balanced, i.e. $|P_1| \neq |P_2|$ - implicitly, one processing unit may have a higher workload than the other. In order to improve the balance, the selection step of the mincut algorithm may prefer balance over optimality of the cut, but this does not improve the degree of parallelism.

Instead, we suggest to execute the mincut algorithm recursively on the individual segmentations basing on two boundary criteria:

1. there exists another segmentation of $P_{1/2}$ so that the partitioning weight of $P_{x.1}$ and $P_{x.2}$ is only slightly higher than that of P_1 and P_2, and more importantly
2. the size $|P_x|$ of the individual partitions, which should meet two aspects:
   a) the results partitions should be roughly balanced, i.e. $|P_1| \approx |P_2| \approx \ldots \approx |P_n|$
   b) the size of the partition should be roughly aligned to the cache sizes of the destination platform

Thereby it is obvious that a partition introducing a higher delay than gain through the introduced parallelism is not worth considering in the first instance. Along that line, the overhead for cache misses may be lower than the communication overhead, so that a larger segment may be preferable over multiple small segments.

Furthermore, as the segmentation may be executed in general, i.e. without a specific destination platform being known, cache size may not be a concrete value that can be used as a boundary criterion. However, it must be kept in mind that segments can be rejoined again at execution time, respectively executed sequentially – this will be discussed in more detail in a future publication.

D) Outlook

The current algorithm takes an initial approach towards offline code segmentation using preliminary heuristics for defining cut-off criteria and boundary conditions. However, these heuristics need to be validated and fine-tuned in some dedicated test runs, as they are planned for the next working period of work package 6 (evaluation). In light of the evaluation definitions specified in the S(o)OS deliverable “D6.1 Evaluation Criteria”, the primary approaches will consist in so-called “abstract” simulations of the algorithm, i.e. with specific functional kernel representatives that show the general same behaviour – this avoids having to disassemble all different kernel and application types. The goal will thereby be to identify the best values for the given heuristics, so that the segmented graph exposes a clear level of concurrency.

In addition to this, the current algorithm does not yet deal with the aspect of hardware requirement derivation, or mapping the requirements and segments to the specifics of the destination platform, as both aspects required more details about hardware specification. This was elaborated in parallel in work package 2, published as part of the deliverable "D2.2 First Implementation Set: Hardware".

The according input will be taken over in the second iteration of work package 4, to establish the algorithms to extract hardware information from the code, as well as to perform a graph matching between the segmented code and the hardware infrastructure description.

Further to this, and as elaborated in the next chapter, we will assess how the segmentation can be refined further through information provided by the user at development time, respectively vice versa, which type of information should be provided in order to improve the segmentation process.

2. Programming Models

The prevailing approach to programming is based on the imperative programming model. That situation is underlying the previous parts of this report, in which the imperative perspective is not discussed explicitly but implicitly assumed. Unmentioned remained the applicative approaches to programming, such as functional programming and (constraint) logic programming, even though in the past expectations from these approaches were reasonably high. Without having the intention to be complete, we mention some possible reasons for the fact that the imperative perspective became the default approach to programming. First of all, it is the type of approach that fits a classical X86 processor architecture since statements in an imperative language are built upon an underlying instruction set in a straightforward way. Secondly, it appeals to an immediate intuition of processing activity where values are taken from memory cells, some operation is performed and the resulting values are stored in memory again.

Given the spectacular development of such architectures over the past decades, it is not surprising that the corresponding imperative programming languages are used extensively. Also, applications became bigger and bigger, and the need for a maintainable structure in software, approaches such as object orientation with encapsulation, multiple inheritance, etc., naturally followed. Now that many applications are written in this way, the existing software has to be maintained, extended, and updated. Hence, the vast majority of programming is performed in the imperative style, and it is only natural that it is silently understood that when discussing programming, imperative programming is meant.

Nevertheless, it is worthwhile to pay attention to other, declarative styles of programming, as recently some trends are visible which are difficult to seal with from an imperative perspective, and which show an increased interest in these approaches again. Reasons for this may be that the spectacular character of the developments of single core processors came to an end (at least for the time being) and the focus shifted to multi/many core systems. That increased the role of concurrency in programming and all the problems that come with that such as deadlock, lost updates, inconsistency, etc. Combined with the huge size of applications, this increased the complexity of programming enormously. One possible approach to tackle this increasing complexity, when sticking to the present day prevailing, i.e., imperative, style of programming, is to develop certain additional mechanisms to protect against errors that are caused by the concurrent character of these programs. Examples of such mechanisms are locks, transactional memory, and extensions of the code with different types of annotations and so called pragmas.

Another possible approach to solving this increasing complexity is to search for a radically different approach to programming, probably on a higher level and with stronger abstraction mechanisms. Such a choice might be adequate given the fundamental character of the problems that arise from the traditional way of dealing with concurrency (see [57]). Besides, it is advocated in papers such as [54][60][61] , and also in some companies (among which Nokia and CapGemini) there are discussions going on in this context.

In this section we will take a look at declarative programming. Since logical languages will not play a role in S(o)OS, we concentrate on programming from a functional perspective and touch upon some points that will be a topic of research in S(o)OS. As some well-known examples of functional programming languages we mention Haskell, Clean and Erlang. LISP and Scheme also fall in this category, though these are hybrid languages. Below we will mostly use Haskell syntax.

The fundamental difference between a functional language and an imperative language is that a functional language only has expressions and no statements. In particular, a functional language has no assignment, where in an imperative language this is an indispensable statement: without assignment one can do nothing. The consequence is that programming is in terms of logical dependencies and mathematical functions, and binding values to names (variables, identifiers), rather than state manipulations by means of actions as prescribed by statements/commands. Hence, while programming, one has no direct memory model in mind, all operations that take place in memory are left to the evaluation system and the operating system.

The downside of course is that this may lead to inefficiency or unpredictability of e.g. memory usage. However, that depends on the way the language and the architecture match each other. Recently some processor architectures are proposed that are good at executing functional programs by exploiting, among other things, the parallelism that is inherently present in functional languages [55] [58][59]. In the context of mastering the complexity of concurrent programming as such we ignore these aspects for now.

Besides, where imperative programming is close to the instruction set architecture of a Von Neumann type of processor, a functional language is close to digital circuits straight away, i.e. the functional perspective is the obvious way to abstract from the architecture on an FPGA or on an ASIC. The reason for that is that a digital circuit transforms input signals into output signals, i.e., a digital circuit can be seen as a function from input signals to output signals. For memory elements in the architecture the same holds: they can be seen as both input signals and output signals. In other words, looking at a digital circuit as a function simply is a mathematical abstraction of a Mealy Machine.

Because of the above indicated functional perspective, functional languages are closer to a mathematical way of understanding than an imperative language. Advantages of that are the immediate availability of polymorphism and higher order functions. Both aspects have great benefits for the ease of programming and the flexibility of expressing algorithms. Besides, expressing algorithms on a level of aggregate data structures promises to give good possibilities for automatic parallelization of algorithms since the operations that might be parallelized are explicitly visible in the program code. To illustrate this feature we return shortly to one of the examples as given in section II.1. A), case 3 (p. 16):

$$A(i)=A(i)+C(i)$$
$$B(i)=B(i)-C(i)+A(i)$$

This example deals with two statements inside a loop that works for all *i* in the range of the arrays *A*, *B*, *C*. The imperative reading here is clearly that memory locations *A*(*i*) and B(i) are updated with the result of the computations at the right hand side. In addition, the first statement is executed before the second one, so the value of $A(i)$ on the right hand side in the first statement is the "old" value, whereas the value of $A(i)$ on the right hand side in the second statement is the "new" value.

As mentioned above, in a functional perspective one does not have a memory model in mind, so variables $A(i)$, $B(i)$, $C(i)$ do not refer to locations, but they have to be considered as variables in a mathematical sense, i.e., they *have* values. In that context one cannot give a different value to the same variable (compare an equation like $x^2 + 2x - 1 = 0$ in which all occurrences of x have the same value). Hence, on the left hand side one has to rename the variables into, say, $A(i)'$ and $B(i)'$. Then one should also replace $A(i)$ by $A(i)'$ on the right hand side in the second line. Note however, that the order in which the "statements" – which in a functional context are termed *definitions* – are executed now is irrelevant: either way the result is the same. In other words, a functional language is inherently parallel. Note also that because of this change in interpretation the inter-dependency between the variables is not influenced by the order of computation, nor by the location where a variable is stored, but purely by a data dependency which is immediately visible in the code itself.

Now, "lifting" these definitions to the aggregate level means that we define the addition operation for arrays (vectors, lists) as pairwise addition on the corresponding elements of the arrays. The same may be done with multiplication, subtraction, etc. Hence, lifting may be considered as a *function*, written as $\hat{\ }$, that works on such (in this case binary) operations. Thus, $\hat{+}$ denotes pairwise addition, $\hat{*}$ pairwaise multiplication, etc.

Note that $\hat{\ }$ is a "higher order function", i.e., it takes an operation as argument and yields an operation as result. Such higher order functions are standard in a functional language and are evaluated according to their mathematical definition.

The above definitions now may be rewritten on the aggregate level (note that on the element level a loop is implicitly implied by this notation):

$$A' = A \mathbin{\hat{+}} C$$
$$B' = B \mathbin{\hat{-}} C \mathbin{\hat{+}} A'$$

For such aggregate representations of the definitions it is a mathematical law that they may be calculated in parallel, i.e., various fragments of the additions may be calculated on different cores, without changing the result.

Other examples of such laws require specific properties of the operations involved, such as associativity, commutativity, etc, in order to be parallelizable. That is to say, the consequence of lifting the definitions to the aggregate level is that one has explicit control over, e.g., automatic parallelization.

Two final aspects of the functional perspective that should be mentioned here are algebraic data types and pattern matching. Without going into details we stipulate that such types are very suitable to express embedded languages so that for example communication primitives, but also services of operating systems, can be expressed as expressions in embedded languages within the framework of the functional perspective. In combination with pattern matching, this gives a concise and comprehensible way of defining such aspects.

A) TOP-DOWN APPROACH TOWARDS PARALLELIZATION

In this section we sketch a high level approach to the partitioning of algorithms, expressed in a data oriented way and using higher order operations. The intention of this section is to search for a way to find parallelizations of for-loops in a "top-down" fashion, using mathematical laws. As a result, the same parallelizations should be found as in the "bottom-up" approach. Below we will sketch the ideas based on an example from [56].

EXAMPLE

In [56] the following simple example is discussed:

```
float f (float scalar)
{
  /* calculation body */
  return r;
}

main (float data)
{
  int i;
  float tmp, carry=data, sum=0;

  for (i=0; i<10; i++) {
    tmp   = f (carry);
    carry = carry + 1;
    sum  += f (tmp);
  }
}
```

Thus, there is an input value data which is given as the initial value to the variable carry. Then a for-loop is executed 10 times, and in each pass through the for-loop the function f is applied to carry, after which carry is increased by 1. Next, f is applied to its previous result (stored in the variable tmp) and added to the accumulative variable sum.

PARALLELIZATION

In [56] the following three possibilities to partition the original algorithm are presented. It's our aim to compare the derivation and formulation of these possibilities with the derivation and formulation of comparable possibilities arising from a more mathematical presentation of the algorithm above, hence we won't discuss the effects of these possibilities, nor their (dis)advantages but just mention them.

Possibility 1 introduces functional parallelism consisting of two threads:

```
  for (i=0; i<10; i++) {
    tmp   = f (carry);
    carry = carry + 1;
    PUT tmp INTO FIFO;
  }

  for (i=0; i<10; i++) {
    GET tmp FROM FIFO;
    sum  += f (tmp);
  }
```

In this alternative the original for-loop is split according to the sequential statement structure inside the loop, respecting the dependency embodied by the variable carry. The dependency embodied by the variable tmp is broken, hence special "fifo-commands" **PUT** and **GET** are introduced to repair this dependency and to take care of the communication between the two for-loops.

Possibility 2 introduces data parallelism, also consisting of two threads:

```
  for (i=0; i<5; i++) {
    tmp   = f (carry);
    carry = carry + 1;
    sum  += f (tmp);
```

```
}

for (i=5; i<10; i++) {
  tmp   = f (carry);
  carry = carry + 1;
  sum  += f (tmp);
}
```
In this alternative the for-loop is split in the first five and the second five of the ten passes through the loop. Note that this does not give rise to the introduction of PUT and GET statements as no dependencies are broken. However, in [56] it is observed of this alternative that it has no positive effect concerning parallelization as both for-loops can only be executed after each other.

Possibility 3 exploits a mix of functional and data parallelism, consisting of three threads:

```
for (i=0; i<10; i++) {
  carry = carry + 1;
  if (i<5)
    PUT carry INTO FIFO1;
  else
    PUT carry INTO FIFO2;
}

for (i=0; i<5; i++) {
  tmp  = f (carry);
  GET carry FROM FIFO1;
  sum += f (tmp);
}
PUT sum INTO FIFO3;

for (i=5; i<10; i++) {
  tmp  = f (carry);
  GET carry FROM FIFO2;
  sum += f (tmp);
}
PUT sum INTO FIFO4;
```
Here the increment of carry is done in a separate for-loop, and the application of f and the accumulation of sum are done in two for-loops, one for the first five iterations, the other for the second five. The second for-loop can start after the first one did put one value of carry in FIFO1, whereas the third for-loop has to wait until a value of carry is put in FIFO2. Note that at the end FIFO3 and FIFO4 still have to be added to together, as it is intended by the authors that duplicated variables are private to each thread.

SEMANTIC APPROACH

Seen in a more semantic or mathematical way, the above algorithm takes ten consecutive values of the input value data (by each time adding 1), then applies f twice to each of these values and takes the sum of the results. Presented in a more mathematical form:

$$\mathrm{sum} = \sum^{9} f\,(f\,(data\, - \tag{1}$$

or, equivalently:

$$\mathrm{sum} = ((0 \oplus) \circ \widehat{f} \circ \widehat{f} \circ \vec{it}\,(+1)\,\mathrm{data})\;10 \tag{2}$$

In this formula the part between brackets is a *function* that is applied to the *list* [0,1,…,9]. This function is a *composition* (indicated by the composition operator $\circ$) of *four* smaller functions, to be applied from right to left. Thus, the first function to be applied to the list [0..9] is

$$\vec{it}(+1)\,\mathrm{data}$$

which yields [*data, data+1, data+2, …, data+9*] by each time adding 1 to the previous result and putting all results in a list. Here, it stands for "*iterate*", and the vector notation $\vec{it}$ means that the function $(+1)$ yields a list (a vector) rather than just applying it 10 times to the initial value *data.* Thus ($g^n(a)$ stands for applying *g n* times to *a*):

$$it\; g\; a\; n = g^n(a)$$

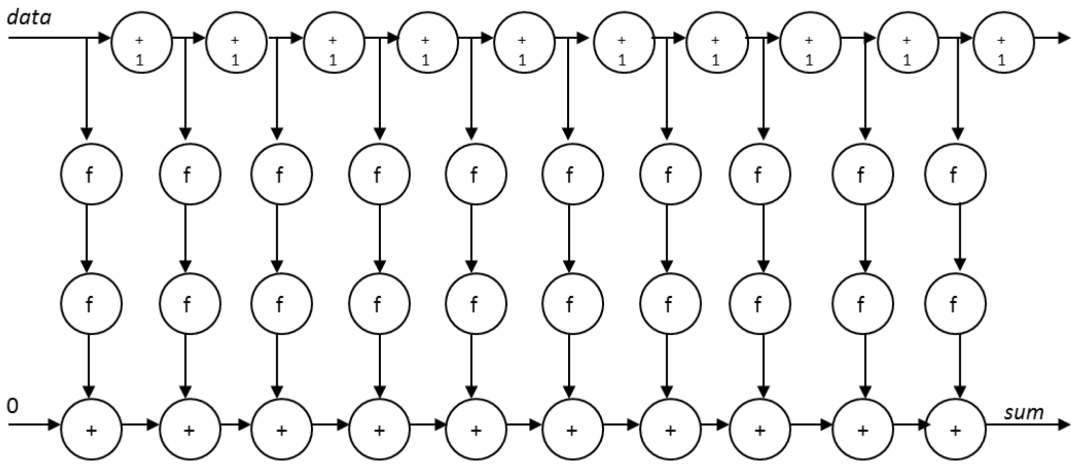
whereas

$$\vec{it}\; g\; a\; n = [a, g(a), \ldots, g^{n-1}(a)]$$

Next, the function $\widehat{f}$ is applied to this resulting list, where the *hat* expresses the *lifting* of the function *f*, i.e., the function *f* is applied to *all* elements in this list. That operation is repeated once more, thus resulting in the list

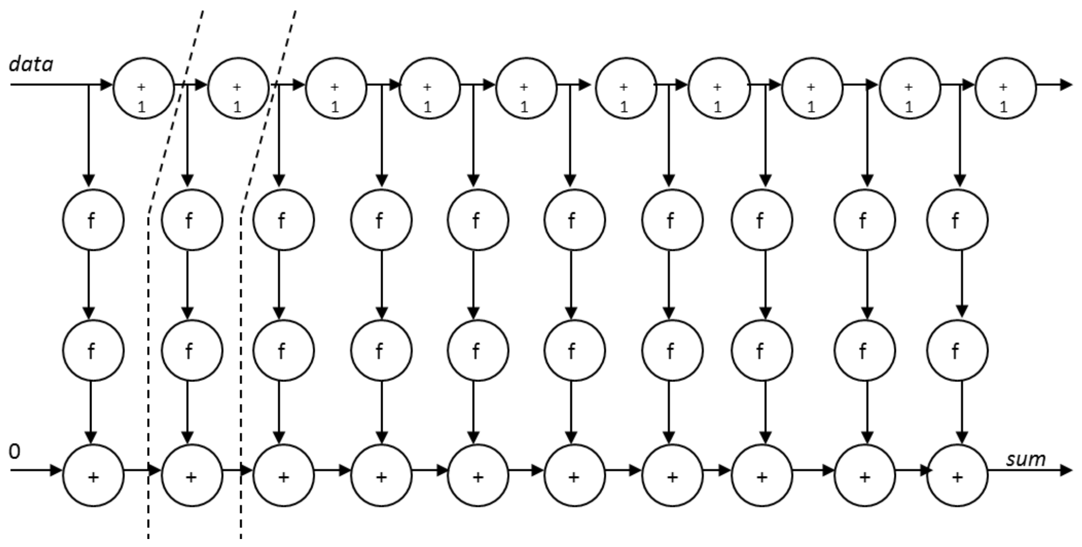
$$[f(f(data)), f(f(data+1)), f(f(data+2)), \ldots, f(f(data+9))]$$

Finally, the function $(0\oplus)$ is applied to this resulting list, meaning that the operation + "rolls over" the list, starting with the value 0. That is to say, all elements in the list are added, starting with 0.



Expressed in a data dependency diagram, formula (2) looks as follows (functions are depicted in circles, and values are on the arrows):

Looking at this dependency diagram, there are several ways of separating the diagram. A first separation is in vertical "slices", where apart from data dependencies all functions in a slice may be executed in parallel:



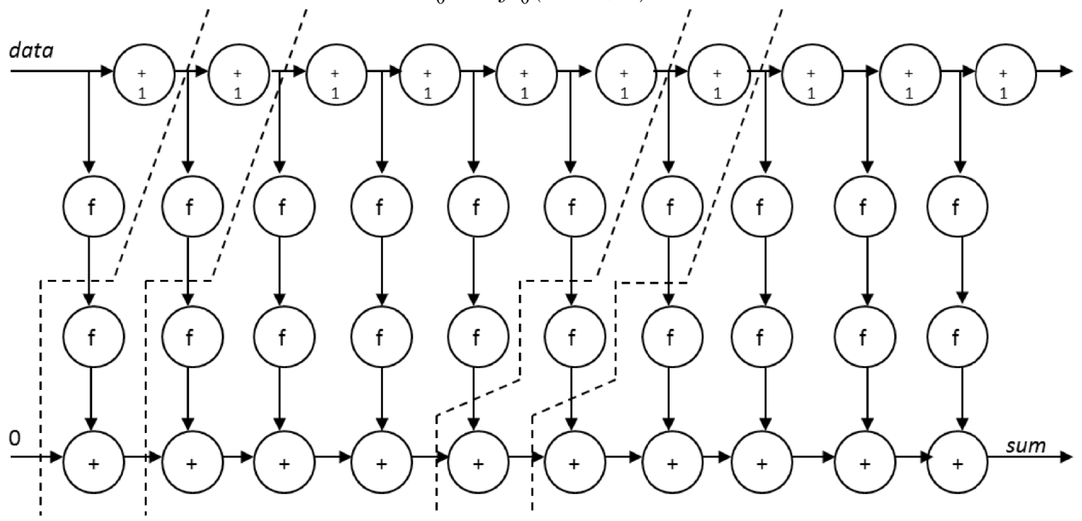
Where data arrows pass through slice borders, values have to be communicated to the resource where the next slice in executed. We remark that *time* is a resource as well, so the above may also be realized by each time executing the functions in a slice on the same core, except in a later period of time. In that case, the communication of the values will be done by memory elements. Hence, this *slicing* coincides with the for-loop in the original algorithm from [56], as given in the beginning of this section.

The slice indicated above contains two parts that may be executed in parallel: the function $(+1)$ that is applied to the *carry* coming in from top left, and the part that consists of applying *f* twice and adding the result to *sum* coming in bottom left. Thus, the definition of this slice f_0 reads as follows (write *c,s* for *carry*, *sum*, respectively):

$$f_0(c,s)=(c',s')$$
$$where\ c'=c+1;\ s'=s+(f\circ f)(c)$$

Now the result res_0 of the whole algorithm is produced by applying the function f_0 ten times, using the *it* function introduced above:

$$res_0=it\ f_0(data,0)\ 10$$



There are other slicings possible, below two examples are given:

Note that in the first of these slicings (f_1) an extra communication item has to be introduced in order to transfer the result of the first application of the function *f* to the second application of *f*, whereas in the second slice in addition the result of the second application of *f* has to be communicated to the +-function. Note also that the first of these two slicings coincides with possibility 1 given above in which the transfer from the first to the second application of *f* is called tmp. Finally, note that the second slice (f_2) is not expressed as one of the possibilities above.

Note that the slices f_1 and f_2 have three and four inputs, respectively. Thus, they can be defined as follows (*c,t,s* stand for *carry*, *tmp*, *sum*, respectively, corresponding to the variable names in the original algorithm):

$$f_1(c,t,s)=(c',t',s')$$
$$where\ c'=c+1;t'=f(c);s'=s+f(t)$$

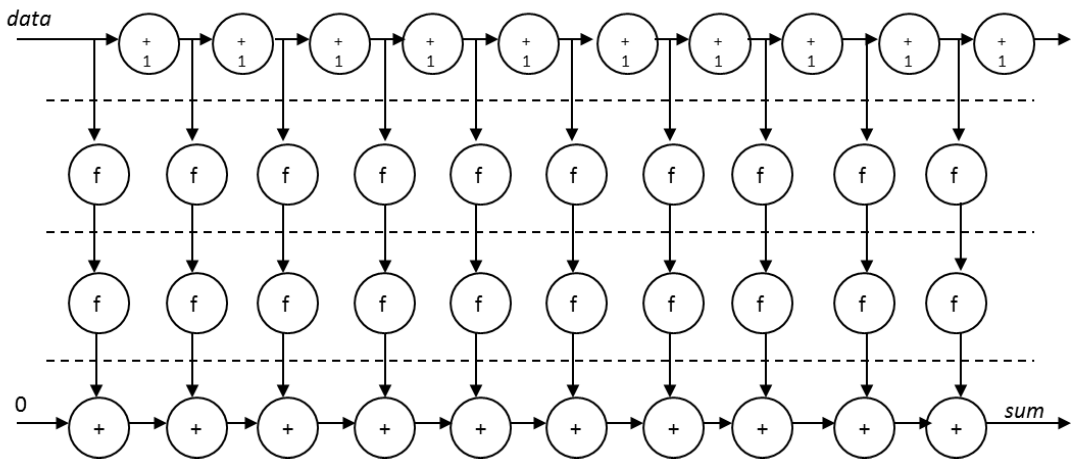
$$f_2(c,t_0,t_1,s)=(c',t_0',t_1',s')$$
$$where\ c'=c+1;t_0'=f(c);t_1'=f(t_0);s'=s+t_1$$

Here too, the final result can be acquired by using the iteration function *it,* though because of the "skew" nature of the slices, some straightforward initialization has to be performed:

$$res_1=it\ f_1(initialize_1(data,0))10$$
$$res_2=it\ f_2(initialize_2(data,0))10$$

In the diagram it can be seen that the functions within the same slice which are not connected by data arrows can be executed in parallel. Thus the second partitioning in the last dependency diagram offers most parallelism of the partitioning given so far: all four functions can be executed in parallel. Clearly, it depends on the complexity of these functions, on the available resources, and on the relevant criteria (efficiency in time, power consumption, etc) which partitioning offers the best results.

As a final example we give the following separation



Clearly, here all instantiations of *f* in a horizontal slice may be executed in parallel, but all the (+1) functions on the top line and the +-functions on the bottom line may not. Hence, this separation is of a different nature and is only adequate if enough resources are available such that indeed all instances of *f* can be executed in parallel. We will not discuss this separation further

Correctness

We will prove the correctness of res_0, relative to *F*, where *F* is derived from the original algorithm as given in equation (2) and generalized as follows:

$$F(x,y)=((x\oplus)\circ\widehat{f}\circ\widehat{f}\circ\vec{it}(+1))\ y$$

In order to prove the correctness of *res₀* we have to prove (see the definition of *res₀*)

$$\pi_2(it\ f_0(y,x)n)=F(x,y)n$$

where π_2 is the projection of a two-tuple to its second element.

We first give the formal definitions of the involved higher order constructions:

$$\begin{gathered}
a\odot[\,]=a\\
a\odot(x:xs)=(a*x)\odot xs\\
\widehat{f}\,[\,]=[\,]\\
\widehat{f}\,(x:xs)=f\ x:\widehat{f}\ xs\\
it\ f\ a\,0=a\\
it\ f\ a\,n=it\ f\ (f\ a)(n-1)\\
\vec{it}\ f\ a\,0=[\,]\\
\vec{it}\ f\ a\,n=a:\vec{it}\ f\ (f\ a)(n-1)
\end{gathered}$$

The proof proceeds by induction on *n*. The case that *n=0* is straightforward:

$$\begin{aligned}
F(x,y)0&=((x\oplus)\circ\widehat{f}\circ\widehat{f}\circ\vec{it}(+1)\,y)0\\
&=x\oplus(\widehat{f}\,(\widehat{f}\,(\vec{it}(+1)\,y\,0)))\\
&=x\oplus(\widehat{f}\,(\widehat{f}\,[\,]))\\
&=x\oplus[\,]\\
&=x
\end{aligned}$$

$$\begin{aligned}
\pi_2(it\ f_0(y,x)0)&=\pi_2(y,x)\\
&=x
\end{aligned}$$

The case for *n+1* goes as follows:

$$\begin{aligned}
F(x,y)(n+1)&=((x\oplus)\circ\widehat{f}\circ\widehat{f}\circ\vec{it}(+1)\,y)(n+1)\\
&=x\oplus(\widehat{f}\,(\widehat{f}\,(\vec{it}(+1)\,y(n+1))))\\
&=x\oplus(\widehat{f}\,(\widehat{f}\,(y:(\vec{it}(+1)(y+1)n))))\\
&=x\oplus(\widehat{f}\,((f\ y):(\widehat{f}\,(\vec{it}(+1)(y+1)n))))\\
&=x\oplus((f\,(f\ y)):(\widehat{f}\,(\widehat{f}\,(\vec{it}(+1)(y+1)n))))\\
&=(x+((f\,(f\ y)))\oplus(\widehat{f}\,(\widehat{f}\,(\vec{it}(+1)(y+1)n))))\\
&=F(x+(f\,(f\ y)),\,y+1)n
\end{aligned}$$

The first step is by the definition of function composition , the second step by the definition of $(+1)$, the third and fourth step by the definition of $\vec{it}$, the fifth step by the definition of O, and the last step by the definition of *F*.

$$\begin{aligned}
\pi_2(it\ f_0(y,x)(n+1))&=\pi_2(it\ f_0(f_0(y,x))n)\\
&=\pi_2(it\ f_0(y+1,x+(f\,(f\ y)))n)
\end{aligned}$$

Here, the first step is by the definition of *it* and the second step is by the definition of *f₀*. Now the proof is completed by the induction hypothesis.

The proofs for *res₁* and *res₂* proceed likewise.

Derivations

In the above the definitions of *fᵢ* and *resᵢ* were motivated by looking at the data dependency diagram of the original algorithm, and by separating the diagram by drawing slices in different ways. We also showed how to prove the correctness of the definitions. However, an important question is whether the above definitions can be *derived* in a formal way from one of the above given mathematical formulations (1) or (2) of the algorithm at hand. In particular, equation (2) seems adequate as a starting point since the usage of the higher order operations $O, \circ, \hat{}, \vec{it}$ gives explicit access to the relevant functions $+, f, (+1)$ and their properties.

For now we only remark that the slices in the diagram above precisely coincide with the function composition operators in the definition of *F*, whereas the horizontal data transitions coincide with the occurrences of the iterative nature of $\vec{it}$ and O. Hence, there is an immediate relationship between equation (2) and the definitions above.

The development of laws to be used by the derivation of the definitions above is currently work in progress and falls outside the scope of this deliverable.

III. Programming Language Extensions

1. Transactional Memory

Programming language extensions are important for supporting transactional memory integration with various programming languages for the following reasons:

1. **Usability.** The compiler and the runtime can perform automatic code instrumentation when the appropriate source-level annotations are used. This relieves the software developer from the task of manually inserting annotations and improves the usability of the transactional memory system as a whole. As an effect, this really brings transactional memory closer to fulfilling its promise of simple concurrent programming for average developers.
2. **Optimizations.** The compiler and the runtime can perform various optimizations to reduce the overheads of the transactional memory operations. Several language extensions can greatly improve the effectiveness of such optimizations, as the developers can provide the required information to the compiler when this information cannot be obtained during the analysis phase.

Some work on the language extensions for supporting transactional memory has already been done. The most notable is the Draft Specification of Transactional Language Constructs for C++ [6], proposed by the Transactional Memory Specification Drafting Group that involves people from several major software companies.

A) Basic Extensions

We first describe several basic extensions of the language and what role they have in improving the usability of transactional memory. We will use pseudocode in the following examples and will use the syntax similar to the C++ transactional language constructs specifications. These and similar constructs are applicable to other programming languages as well.

In order to use the transactional memory, the software developer needs to transform the code in the following way:

1. Insert calls to transactional memory library to denote the start and the end of the transaction. In our examples we will use `tx_start()` and `tx_commit()` calls for this purpose.
2. Replace all accesses to shared variables from inside transaction by appropriate calls to transactional memory library. This means that all read accesses to shared memory locations have to be replaced with `tx_read(addr)` calls and that all write accesses to shared memory locations have to be accessed with `tx_write(addr, val)` calls.
3. Perform the instrumentation from previous step for all functions that are invoked from transactional blocks or other functions invoked from transactional blocks (recursively).

In the following example, we demonstrate the application of these rules to a simple code segment:

```
atomic {
   x = x - 20;
   y = y + 20;
   fun(x, y);
}
```

```
fun(x, y) {
   z = z + x - y;
}
```

After the transformations the code segment becomes:

```
start_tx();
tx_write(x, tx_read(x) – 20);
tx_write(y, tx_read(y) + 20);
fun_tx(x, y);
commit_tx();

fun_tx(x, y) {
   tx_write(z, tx_read(z) + tx_read(x) - tx_read(y));
}
```

The instrumented code shows the previously listed steps and, in addition, it also shows that a transactional version of a functions is created for all the functions that can be called from any transaction. This is necessary in order for the programmer to be able to invoke the regular, non-transactional, versions of the functions from non-transactional code when necessary.

It is obvious that these simple transformations can be done by the compiler. Next, we will briefly describe the basic set of programming language extensions that are required by the compiler:

1. **Atomic blocks.** Atomic blocks are used to denote the actual transactions in code:

   ```
   atomic {
      a = a + 1;
      b = b + 1;
   }
   ```

   Atomic blocks can be nested as well, in which case the transactional memory system has to guarantee the correct semantics. Here, we assume that transactional memory guarantees closed transaction nesting semantics.

2. **Transaction callable functions.** Functions that have to be callable from transactions should be annotated to allow the compiler to generate their transactional versions. In many cases, the function annotation is not necessary as it can be inferred at compile time, but in some cases the inference is impossible. In those cases, the annotation is required or the compiler or the runtime will generate an error.

   ```
   tx_callable fun(x, y);
   ```

3. **Transaction non-callable functions.** Functions that cannot be called from transactions (e.g. because they have some side-effects that transactional memory cannot roll back) also need to be annotated as such.

   ```
   tx_non_callable write_to_disk(file, data);
   ```

4. **Transactional functions.** In some cases, functions that are implicitly invoked (e.g. constructors in C++) have to be executed transactionally because they access shared data. To ensure that these functions start transactions if they are not invoked from inside one, they should be annotated as well.

   ```
   transactional fun(x, y, z);
   ```

B) Optimization extensions

There are many cases in which the compiler or the runtime perform the instrumentation of memory accesses because they conservatively assume that accessed memory is shared (this is needed for ensuring correctness) when the memory is actually local to the thread or to the transaction. The programmer can help the compiler and the runtime in these cases by using some of the language extensions to eliminate the unnecessary instrumentation. Invoking transactional memory functions to perform reads and writes of shared data is much more expensive than simply doing so with a single CPU instruction. For this reason it is very beneficial to remove the unnecessary instrumentation.

Some of the language extensions proposed in this section are more applicable to weakly-typed languages or languages that allow explicit type conventions (such as C and C++), because in these languages the extent of analysis that can be performed statically is limited. For example, in C++ it is possible to declare a constant pointer and have it point to the non-constant variable. This prevents the compiler from applying some of the optimizations without the help from the programmer.

1. **Side-effect free functions.** Some functions are side-effect free and there is no need to perform transactional accesses inside these functions. A typical example of side-effect free functions are functions for calculating values of mathematical functions, such as sin(x), cos(x), etc. The compiler cannot usually identify these functions correctly and requires the help from the developers for this.

```
tx_pure sin(x);
tx_pure cos(x);
```

2. **Thread-local data and functions.** Thread-local data is data that is accessed only by the current thread. Some data can alternatively be thread-local and shared, depending on the particular application. The compiler cannot distinguish thread-local from shared data in all cases. Therefore it has to conservatively treat data as shared if it cannot determine that it is thread-local for sure. The programmer can help the compiler by annotating data as thread-local. Furthermore, some functions could be accessing strictly thread-local data, in which case no instrumentation in the functions is required at all.

```
// declaring data thread-local
thread_local x;
thread_local y;

// declaring functions thread-local
thread_local fun(x, y, z);

// declaring function parameters thread-local
fun(x, z, thread_local u);
```

3. **Transaction-local data and functions.** Similarly to thread-local data and functions, some data is accessed only by a particular transaction. This typically occurs when the data are allocated inside a transaction and subsequently accessed. The compiler cannot always propagate the information about the allocated data across function calls and flow control structures. In these cases it requires help from the programmers to optimize away the unnecessary transactional accesses.

```
// declaring data transaction-local
tx_local x;
tx_local y;

// declaring functions transaction-local
tx_local fun(x, y, z);
```

```
// declaring function parameters transaction-local
fun(x, z, tx_local u);
```

4. **Constant data.** Similarly to thread-local and transaction-local data, some data is constant during parts of, or the whole, execution of the application. As discussed above, it is not always possible for the compiler to identify the constant data and optimize the transactional accesses accordingly. For this reason, the compiler might require the annotations by the programmer in order to reduce the instrumentation overheads as much as possible. It is interesting to note that, even in some strongly typed languages that do not allow as much explicit conversions as C/C++ it can be difficult to identify the constant data. In particular, if some data is updated during one phase of the application (e.g. during initialization) and becomes constant only later, the compiler requires annotations by the programmer in order to perform appropriate optimizations.

```
// declaring data constant
tx_const x;
tx_const y;

// declaring functions constant
tx_const fun(x, y, z);

// declaring function parameters constant
fun(x, z, tx_const u);
```

5. **Semantic conflict detection.** Semantic conflict detection is a promising approach to improving performance and scalability of transactional memory applications. With semantic conflict detection, the conflict detection is not done at the level of each accessed memory location. Instead, it is done based on the semantics of the accessed object. As an example, if some data structure is used in the application with the semantics of the set of values, then the conflict detection can benefit from commutativity of operations that do not access the same element in the set. This typically significantly reduces the overheads of the accesses to the set as well as the transaction abort rate, thus improving both the performance and the scalability of the applications. One prominent example of semantic conflict detection is Transaction Boosting [7].

   The downside of the semantic conflict detection is that it requires careful implementation of underlying concurrent data structures (in our example the data structure that is used to implement the set) and careful design of the semantic conflict detection algorithm. These downsides are alleviated by the existing efficient concurrent data structures algorithms and implementations.

   The compiler needs to enable two different mechanisms for the semantic conflict detection to be seamlessly integrated into the underlying language:

   a) **Semantic transaction functions and data types.** Similarly to previously described optimization annotations, this annotation also signals to the compiler that the instrumentation of some functions and data types is not needed. We prefer to introduce a separate annotation for each of the cases in order to make it declaratively clear why the instrumentation can be suppressed and also for the cases when handling of different optimizations differs slightly. (Alternatively, all the annotations could be merged into one that supports the superset of annotations' semantics for ease of development.)

```
// declaring semantic transaction function
tx_semantic fun1(x, y);
tx_semantic fun2(x, y, z);
```

```
// declaring semantic transaction types
tx_semantic class Class {
fun1(x, y);
fun(x, z, tx_const u);
};
```

The assumption here is that the developer can use the transactional memory API to implement the appropriate semantic conflict detection without any further support from the compiler. Alternatively, further language extensions could be defined to specify internal semantic conflict detection mechanisms and the way they are accessed.

b) **Compensating actions.** One special case where the compiler can further improve handling of semantic conflict detection are the compensating actions. The objects that use semantic conflict detections typically require some compensating actions to be executed if the transaction is aborted due to some conflicts. (In our set example, the implementation of the set insert function would register the handler to remove the freshly inserted element on transaction abort.) These compensating actions can be specified at the library API level, but it can be clearer and more convenient to specify them using the language extension. A simple syntax to enable declaration of compensating actions enables specifying several different compensating action blocks that all execute when the transactions in which they are executed aborts.

```
atomic {
tx_semantic {
ptr = alloc(size);
} abort_action {
free(ptr);
}
tx_semantic {
fun2(x, y);
} abort_action {
fun2_inverse(x, y);
fun3(x);
}
x++;
}
```

In this example, if the declared function aborts at the last operation (`x++`), then both specified abort actions will be executed with the values of variables they had at the time the preceding semantic conflict detection block finished execution.

C) MESSAGE PASSING TRANSACTIONAL MEMORY LANGUAGE EXTENSIONS

Some message passing transactional memories use an approach that resembles the client-server distributed systems, but keep both the clients and the servers inside the same machine. This approach benefits from light-weight message passing between clients and servers located on the same machine (as discussed in WP 3.2).

For such transactional memory designs, required language extensions are similar to what we already discussed (e.g. the extensions used for instrumentation of transactional memory accesses and functions). However, some additional extensions are needed to facilitate client-server nature of the system organization. These extensions can both make the systems easier to use by the programmers and can be used to perform some simple optimization of the communication protocol and instrumentation.

1. **Server object interface.** Language extensions for defining server objects interfaces are needed. These can take the form of traditional IDL languages or, more appropriately, the form of extensions of the host language.

```
server class Server {
fun1(x, y);
fun2(x, y, z);
}
```

2. **Type sizes.** Using known sizes for objects that are exchanged between the clients and the servers can often be used to optimize the message passing protocol. Sometimes, the object sizes can be inferred by the compiler, but sometimes they cannot. It is usually enough to specify the maximum size of the parameters passed in function invocation. We envision that it would be enough to distinguish between the small and the large messages, where the size of the small messages would be set to 32 or 48 bytes, depending on the message passing library used. In essence, we expect that the size of the small message would be cache line size without all the required headers used for message passing and RPC headers.

```
class Server {
     small_param fun1(x, y);
     fun2(x, y); // inferred by the compiler
  }
```

2. Real-Time

Developing real-time applications needs always a special attention by developers due to the critical importance that timing constraints, deadlines and generally performance-related metrics possess in this kind of applications. The problem of meeting such kind of application-specific timing constraints needs sometimes special features to be available in the runtime environment the application runs within, such as special scheduling policies or configurations. When such features are present and available in the underlying Operating System, the problem of exploiting properly them is sometimes simplified thanks to the use of proper software components, such as well-designed interface libraries or distributed middleware elements. In the following, we make a short overview of the real-time programming interfaces available for real-time computing nowadays on general-purpose operating systems in A). This is to be considered a compendium to the S(o)OS Deliverable D5.1 [39], which is necessary before introducing in B), C) and D) a few of the innovations that we are working on for this important class of applications, in the context of the programmability of real-time applications on multi- and many-core systems, and the possibilities to correctly analyze their capabilities to meet the associated timing requirements.

A) Programming Interfaces for Real-Time Computing

New generation interactive distributed applications, such as various multimedia, virtual collaboration and e-learning applications, have significant demands on processing, storage and networking capabilities, as well as stringent timing requirements. For these applications, the time at which individual computations and data transfers are terminated is as important as their functional correctness. Contrarily to the traditional domain of hard real-time systems, where a single violation of the timing constraints is not acceptable because it would lead to potential overall system failures and/or life losses, the mentioned applications posses instead *soft* timing constraints, whose violation leads to degradation of the offered Quality of Service and interactivity level.

Soft real-time applications, and especially multimedia ones, would greatly benefit from a real-time run-time support like commonly available on Real-Time Operating Systems. In fact, this type of OS usually provides those features that allow for a well-known, predictable and analyzable timing behaviour of hosted applications: all kernel segments are characterized with well-known worst-case durations, scheduling latencies and interrupt latencies which may be controlled through the appropriate tuning of the interrupt and process schedulers. A set of real-time scheduling policies is available for the system designer and programmers. Time may be measured (and timers may fire) with a high precision (typically sub-millisecond) while there are tools available for WCET estimation and (off-line) analysis, as well as for schedulability analysis. Unfortunately, such OSes are designed for embedded control applications, thus they imply serious constraints on the supported hardware and available high-level software infrastructures.

Nowadays multimedia applications are increasingly complex and they tend to be distributed, thus their development on a hard real-time OS may be overly prohibitive, due to the lack of such OS functionality as: compression libraries, the support for a wide range of multimedia devices and protocols, including the possibility to handle various types of media, and the availability of a complete networking stack.

On the other hand, General-Purpose Operating Systems (GPOS) constitute the ideal development platform for multimedia applications. Unfortunately, GPOSes are not designed to provide the run-time support that is necessary for meeting timing requirements of individual applications. Therefore, in recent years, various efforts have been done towards the integration of real-time technologies within GPOSes, particularly within Linux, for the availability of its kernel as open-source and its worldwide diffusion. Various drawbacks make the Linux kernel, as well as most of the GPOSes, particularly unsuitable for running real-time applications: the monolithic structure of the kernel and the wide variety of drivers that may be loaded within, the impossibility to keep under control all the non-preemptable sections possibly added by such drivers, the general structure of the interrupt management core framework that privileges portability with respect to latencies, and others. As a result, the latency experienced by time-sensitive activities can be as large as tens or hundreds of milliseconds. This makes Linux non-suitable for hard real-time applications with tight timing constraints. Though, soft real-time applications may run quite well within such an environment, especially when the original kernel is modified so to integrate the necessary real-time capabilities.

In what follows we make an overview of the approaches that aim to integrate real-time capabilities within the Linux kernel.

POSIX

POSIX stands for Portable Operating System Interface [34] and is the collective name of a family of standards, also referred as IEEE 1003 or ISO/IEC 9945, jointly developed by the IEEE Portable Application Standards Committee (PASC) and the Austin Common Standards Revision Group (CSRG) of The Open Group.

In 1998, the first real-time profile, IEEE Std 1003.13-1998, was published enabling POSIX to address real-time applications, even for embedded systems and small footprint devices. It must be said that support for most of these functionalities is not mandatory in a POSIX-conforming implementation and the 1003.1-2001 standard defines the X/Open System Interface (XSI) extensions that groups together several of these optional features in the so-called XSI Option Groups. A compliant XSI implementation has to support at least the following options: file synchronization, memory mapped files, memory protection, threads, thread synchronization, thread stack address attribute and size, and may also support a bunch of other option groups among whom: Realtime (grouping together asynchronous, synchronized and prioritized I/O, shared memory objects, process and range based memory locking, semaphores, timers, realtime signals, message passing and process scheduling), Advanced Realtime (grouping together clock selection, process CPU-time clocks, monotonic clock, timeouts and typed memory objects), Realtime Threads (grouping together thread priority inheritance and protection, and thread scheduling), Advanced Realtime Threads (grouping together thread CPU-time clocks, thread sporadic server, spin locks and barriers).

REAL-TIME SCHEDULING SUPPORT IN THE LINUX KERNEL

Linus Torvalds, since the 1990s, coded the Linux kernel to be as POSIX compliant as possible, although he did not own a copy of the standards at that time and based the system call behaviors on what he can read on the man pages of existing UNIX systems. Actually, the Linux Standard Base (LSB) exists as an attempt to standardize Linux-based systems for increased compatibility, it is based on POSIX specifications, Single UNIX Specification and other open standards and is de-facto accepted and followed by almost all GNU/Linux distributions.

The Linux OS implements many of the real-time extensions to POSIX, but at the moment it is lacking the important scheduling mechanisms that allow for realizing temporal isolation among applications (i.e., the SCHED_SPORADIC scheduling class).

Although not being a real-time system, the 2.6 Linux kernel includes a set of features making it particularly suitable for soft real-time applications. First, it is a fully preemptable kernel, like most of the existing real-time operating systems, and a lot of effort has been spent on reducing the length of non-preemptable sections (the major source of kernel latencies). It is noteworthy that the 2.6 kernel series introduced a new scheduler with a bounded execution time, resulting in a highly decreased scheduling latency. Also, in the latest kernel series, a modular framework has been introduced that will possibly allow for an easier integration of other scheduling policies. Second, although being a general-purpose time-sharing kernel, it includes such real-time extensions to the POSIX standard [19] as the real-time POSIX extensions related to signals and timers, and the SCHED_FIFO and SCHED_RR process scheduling policies, enriched with the priority inheritance protocol for avoiding the well-known priority inversion problem.

These policies allow the programmer to specify the real-time priority at which a process may request to be scheduled. All other non real-time processes in the system are scheduled in the background of the real-time processes, allowing real-time processes to exhibit a far better responsiveness than non real-time ones. However, the actual guarantees a real-time process receives depend basically on what other real-time processes run into the system at a higher real-time priority. The SCHED_FIFO and SCHED_RR POSIX scheduling classes have the heavy drawback that a higher priority process can indefinitely delay the execution of all other lower priority ones, and there is not run-time mechanism that ensures temporal encapsulation among real-time processes.

Such problem is mitigated by the SCHED_SPORADIC POSIX real-time scheduling class. In this model, the developer may reserve the CPU for a real-time task by specifying a *budget*, a *period*, a *real-time priority* and a *low priority*. The kernel runs the process at the specified real-time priority for an overall time duration not exceeding the budget in each time window as wide as the period. When the budget is exhausted, the process is downgraded to the low priority. The advantage of such a scheme is that, with proper refinements [21][22], it is possible to rely on classical Fixed-Priority analysis for checking schedulability of the real-time processes on the system. Also, if an application tries to consume more CPU than the maximum value theoretically foreseen in the analysis, i.e., the reservation budget, it is downgraded to the low real-time priority. Therefore, it is possible to keep under control the maximum interference that a higher priority real-time process may have on lower priority ones. The SCHED_SPORADIC POSIX scheduling class has been implemented in the Linux kernel in the context of the IRMOS European Project by Faggioli et al. [23] and made available as a kernel patch [30].

When using Fixed-priority scheduling, the optimum assignment of priorities is the well-known Rate-Monotonic, which needs knowledge of the entire set of real-time tasks running on the system, in order to set each one priority. Interfaces that are more suitable to open, dynamic real-time systems are based on EDF scheduling and they allow each application to request to the OS scheduling guarantees in terms of a minimum budget to be granted every application period. This model is also capable of theoretically saturating a single-processor system with real-time tasks, conversely to what happens with Fixed-priority scheduling.

For example, such an API has been developed for Linux in the context of various academic projects, such as the ones summarized in what follows.

OCERA AND FRESCOR

Linux/OCERA is a variant of the Linux kernel developed as a result of the OCERA (Open Components for Embedded Real-time Applications) European project [31] that embeds within the Linux OS itself a reservation-based scheduling policy [26] typical of real-time systems. This way, standard Linux applications, making use of the full set of resources available through OS services and libraries, may at the same time exploit the benefits of predictable timing behavior and temporal isolation for those threads that are most computation intensive. Actually, OCERA provided a customized distribution of the Linux kernel suitable for both hard and soft real-time activities, that integrates: RT-Linux, a set of drivers useful in the context of industrial control (i.e., for the CAN Bus), Linux with soft real-time extensions at the scheduler level enriched with a feedback-based QoS controller useful for continuous adaptation of the scheduling parameters to the actual task requirements. A unified kernel configuration interface allows one to decide what components to compile in the OCERA kernel.

An approach similar to OCERA has also been adopted by Linux/RT, a commercial variant of Linux supported by TimeSys Inc. and based upon the original Linux Resource Kernel (Linux/RK) from Carnegie Mellon University [27]. In Linux/RT the kernel has been directly modified so to provide CPU, network and disk reservations directly to user processes. This allows the provision of timing guarantees to legacy Linux applications in a transparent way. Moreover, it is possible to access a specific API to take advantage of the reservations and of quality of service management facilities.

The OCERA soft real-time scheduler inside the Linux kernel has been almost rewritten within the FRESCOR (Framework for Real-time Embedded Systems based on COntRacts) European project [32], also due to the main changes undergone by the kernel from the 2.4 to the 2.6 series, resulting in the AQuoSA framework, described below. Furthermore, the FRESCOR project focused on development of components for supporting distributed and multi-resource applications, and realizing more complex QoS control strategies that also account for power-scaling issues typical of embedded applications, and that are also suitable for distributed embedded applications. In fact, as of now, in FRESCOR there are kernel components for providing resource reservation capabilities at the CPU, disk and networking level, that may be accessed through a uniform API designed around the POSIX well-known real-time extensions API. Also, hard real-time components are being developed for hard real-time OSes, in the project.

Generally, the main objective of the FRESCOR project was to develop the enabling technology and infrastructure required to effectively use the most advanced techniques developed for real-time applications with flexible scheduling requirements, in embedded systems design methodologies and tools, providing the necessary elements to target reconfigurable processing modules and reconfigurable distributed architectures.

The approach integrates advanced flexible scheduling techniques directly into an embedded systems design methodology, covering all the levels involved in the implementation, from the OS primitives, through the middleware, up to the application level. This is achieved through a contract model that specifies which are the application requirements with respect to the flexible use of the processing resources in the system, and also what are the resources that must be guaranteed if the component is to be installed into the system, and how the system can distribute any spare capacity that it has, to achieve the highest usage of the available resources.

In the context of IRMOS, the main disadvantage of FRESCOR is its focus on embedded systems, so, for example, multiprocessor capabilities of the platform are only marginally addressed, and the Linux components developed at the scheduler level do not support it.

ADAPTIVE QUALITY OF SERVICE ARCHITECTURE (AQUOSA)

The AQuoSA framework [25] enhances a standard GNU/Linux system with scheduling strategies based on the Resource Reservation techniques. AQuoSA is designed with a layered architecture as depicted in the following figure.

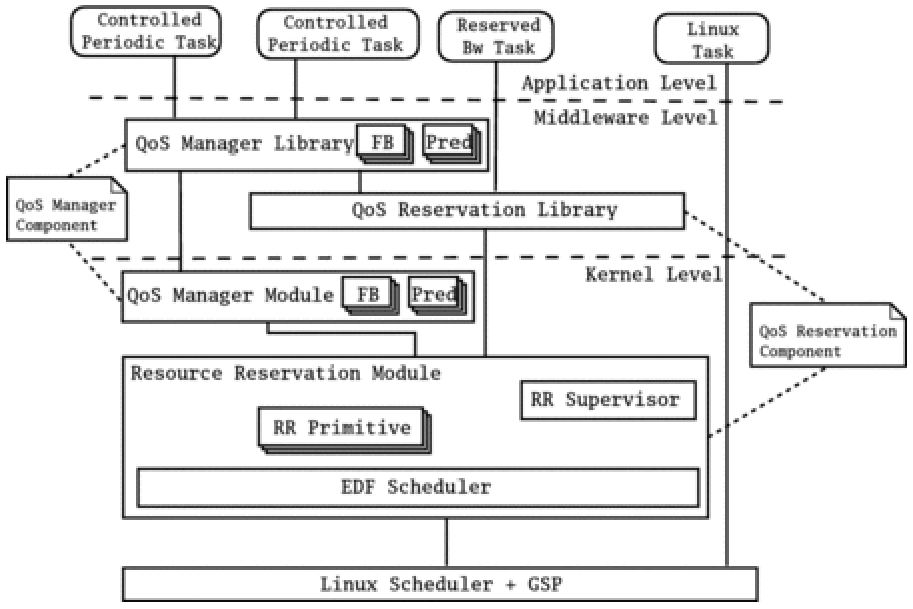


Figure 9: The AQuoSA Architecture

At the lowest level, there is a small patch (Generic Scheduler Patch, GSP) to the Linux kernel that allows dynamically loaded modules to customize the CPU scheduler behaviour, by intercepting and reacting to scheduling-related events such as: creation and destruction of tasks, blocking and unblocking of tasks on synchronization primitives, receive by tasks of the special SIGSTOP and SIGCONT signals). A Kernel Abstraction Layer (KAL) aims at abstracting the higher layers from the very low-level details of the Linux kernel, by providing a set of C functions and macros that abstract the needed kernel functionalities. The Resource Reservation layer (RRES) implements a variant of the CBS scheduling policy on top of an internal EDF scheduler. The QoS Manager layer allows applications to take advantage of Adaptive Reservations, and includes a set of bandwidth controllers that can be used to continuously adapt the budget of a resource reservation according to what an application needs. A well designed user-space library layer allows extending standard Linux applications to use the AQuoSA functionality without any further restriction imposed on them by the architecture. Thanks to an appropriately designed access control model [28], AQuoSA is available to non-privileged users under a security policy that may be configured by the system administrator, not only to privileged users (as it happens for other real-time extensions to Linux). An interesting feature of the AQuoSA architecture is that it does not replace the default Linux scheduler, but coexists with it, giving to soft real-time tasks a higher priority than any non-real-time Linux task. Furthermore, the AQuoSA architecture follows a non-intrusive approach by keeping at the bare minimum (the GSP patch) the modifications needed to the Linux kernel. Unfortunately, in the current version, AQuoSA only supports single-processor systems.

THE IRMOS REAL-TIME SCHEDULER

In the context of the IRMOS European Project[3] (Interactive Real-Time Applications on Service-Oriented Infrastructures), a new real-time scheduler for Linux has been developed by the Real-Time Systems Laboratory of Scuola Superiore Sant'Anna in Pisa. In the following, a general overview of this new scheduler is made, describing its features and how it can be practically used. For the interested reader, more details can be found in [35][36][37].

The IRMOS realtime scheduler (a.k.a., EDF throttling or realtime throttling), allows the administrator to reserve a "slice" of the processing capability of a system for a group of Linux tasks. It is based on a direct modification of the POSIX realtime scheduling class within the Linux kernel, and in particular, to the throttling mechanism already built into the kernel for realtime tasks. Basically, the realtime throttling mechanism is changed from a mechanism that *exclusively limits* the computation power granted to groups of realtime tasks, to one that provides them with both a limit and precise scheduling guarantees (in terms of a guaranteed runtime every period, on each of the available CPUs). Also, it has been designed from scratch with SMP support in mind, and it implements a hierarchical scheduling policy based on both deadlines and priorities. Specifically, POSIX fixed priority (FP) realtime scheduling is nested inside EDF (Earliest Deadline First) scheduling.

The IRMOS realtime scheduler allows for the provisioning of scheduling guarantees to individual task groups. This provisioning is done by specifying two scheduling parameters: a *budget* Q and a *period* P. The tasks in the group are entitled to run on each of the CPUs (processor, or cores when present) available on the platform, for Q time units every period of P time units. This constitutes a scheduling guarantee and a limitation at the same time.

For example, on a single-CPU system, a single task attached to a reservation of 10ms every 100ms is guaranteed to be scheduled on the CPU for 10ms every 100ms. If the task tries to execute for more than 10ms, then the scheduler removes it from the run queue until the next period, at which point its budget is refilled. So if the system has no other ready tasks to schedule, then the CPU goes idle in this time.

Note that periods of different reservations may be specified independently from each other, and the above guarantee is still valid.

The EDF-based scheduler applies a simple admission control rule that decides whether a new reservation may be accepted; it works by ensuring that the sum of the utilizations (budget over period) of all the reservations is less than or equal to the maximum configured share assigned to realtime tasks. This limit may be configured through the `cpu.rt_runtime_us` and `cpu.rt_period_us` entries of the root-level cgroup filesystem (see below). Theoretically, the EDF-based scheduling algorithm allows for full utilization of each CPU by realtime tasks, provided that those tasks can be properly partitioned across the CPUs. However, from a practical perspective, this is far from being a desirable working condition.

Hierarchical Scheduling

The IRMOS scheduler features hierarchical scheduling, mixing both deadline-based and priority-based scheduling. Specifically, POSIX priority-based realtime scheduling is nested inside EDF-based scheduling. The situation is depicted in the figure to the right.

When a reservation is selected to run by the partitioned EDF-based scheduler, a global POSIX priority-based scheduling policy decides what tasks belonging to that reservation will actually run on each CPU. If there are M CPUs, at most the M tasks with the highest priority (among the ones belonging to the reservation group) are the ones which actually run. The system performs admission

3 More information is available at: http://www.irmosproject.eu/.

control over admitted reserved groups, so that the overall system capacity may be properly partitioned among concurrently running activities in the system, without overloading it. Also, the scheduler has a hierarchical configuration capability, by which it is possible to define groups and nested subgroups of realtime tasks with given scheduling parameters.

Admission control and deadline guarantees

When considering realtime systems, we might be concerned about how, exactly, to exploit the described realtime scheduling policy in order to provide proper realtime guarantees to applications. In relatively simple cases, the answer is straightforward. For example, a classical periodic realtime task with a known worst case execution time (WCET) of C and minimum inter-arrival period of T, can be scheduled within a reservation with budget equal to C and reservation period equal to T and will not miss any deadlines. Also, the admission test in this case is the well-known Liu and Layland test for EDF realtime tasks (sum of utilizations must be less than or equal to 1).

However, looking at realtime theory, one easily finds much more complex realtime task models, which include activation offsets, maximum blocking times, durations of critical sections accessing shared resources, etc. Also, real-world realtime applications are often complex multi-threaded applications (think of `vlc`) which are very far from behaving like foreseen by the "ideal" periodic or sporadic task model, and whose activation times are driven by disk I/O and networking instead of (or in addition to) timers. Furthermore, if the application is distributed, one has usually a distributed end-to-end deadline constraint to deal with, something out of reach for a kernel-level task scheduler.

Under such a challenging scenario, it is still possible to schedule realtime applications with a simple policy based on the fundamental principle of temporal isolation, like the one being presented in this section, and provide the necessary guarantees. However, the admission test becomes complex, involving long and involved computations, thus prohibitive for the kernel.

This complexity is why, in the EDF-based scheduler described above, the realtime scheduling parameters communicated to the kernel are kept at the bare minimum and are used in a very simple admission control test. This approach does not try to guarantee that all admitted applications will meet their deadlines, but rather it aims to provide to each application a guaranteed share of the available underlying computing power, with a precise timing granularity. Whether this is sufficient or not for guaranteeing the performance of specific applications must be confirmed by other means, involving a proper design methodology and benchmarking process, possibly with the help of user-space middleware.

Userspace Interface

In order to use the realtime scheduler capabilities, the Linux cgroup filesystem needs to be mounted. This can be achieved with this sequence of commands:

```
mkdir /cg
mount -t cgroup -o cpu,cpuacct cgroup /cg
```

The Linux cgroup filesystem [38] is a mechanism for communications among userspace applications and the kernel based on a filesystem interface. When writing to special filesystem entries, applications can provide information to the kernel. Similarly, when reading the kernel can provide information to the applications. The “cpu” controller of the cgroup filesystem provides the interface to the scheduler, allowing customisations of the scheduling policy and parameters.

By default, up to 95% of the CPU power is allocated to standard POSIX realtime tasks in the root group, which doesn't leave much left over for reservations. So, before we can create a new group, we need to reduce the runtime for root-level tasks, e.g., lowering it to 200ms every 1s:

```
    echo 200000 > /cg/cpu.rt_rt_task_runtime_us
```

Now we can create a new group, with a reservation of 10ms every 100ms:

```
    mkdir /cg/g1
    echo 100000 > /cg/g1/cpu.rt_period_us
    echo 10000 > /cg/g1/cpu.rt_runtime_us
    echo 100000 > /cg/g1/cpu.rt_task_period_us
    echo 10000 > /cg/g1/cpu.rt_task_runtime_us
```

At this point, the new group has no associated tasks. We can attach a task by writing its Linux thread id (tid) to the `tasks` special file entry available in the group folder:

```
    echo 1421 > /cg/g1/tasks
```

Now the attached task has only been added to the group, but it still has its own scheduling class, defaulting to `SCHED_OTHER`. In order to exploit realtime scheduling, we need to assign to the task one of the realtime classes and a realtime priority:

```
    chrt -r -p 20 1421
```

At this point, the task is running with the configured scheduling guarantee (and limitation) of 10ms every 100ms.

Usability and AQuoSA integration

As shown above, the interface towards the new realtime scheduling functionality is based on the cgroup filesystem. While constituting a perfectly usable interface for scripting languages and system administrators, this kind of interface makes programming realtime applications which exploit the new scheduler functionality quite cumbersome: in order to create new reservations, folders need to be created in the cgroup filesystem; for setting scheduling parameters, numbers need to be formatted and written to cgroup entries; for reading them, cgroup entries need to be read back and parsed; etc. The libcgroup library[4] may be of some help for such issues, but it carries non-negligible overhead into the applications. This may be especially troublesome for adaptive applications, e.g., multimedia ones, that might need to change dynamically the reservation parameters following the dynamic workload.

Furthermore, when changing both scheduling parameters (runtime and period), operations need to be carried out in a proper order which depends on the previous values of the parameters themselves, otherwise the admission control logic may reject one of the intermediate steps. Also, while playing with the scheduling parameters (e.g., while tuning the application's performance), one is forced to use intermediate configurations which are highly undesirable. For example, reducing both the budget and the period by an order of magnitude, such as from `(100,200)` to `(10,20)`, one needs to reduce the runtime first, obtaining `(10,200)`, then the period. However, in the intermediate configuration the realtime task is likely to fail due to the insufficient resources being granted.

Also, in the future, the number of parameters needed to configure the realtime scheduler's behavior is expected to (slightly) grow. What is needed from an application development perspective, is an atomic way of setting and changing them, possibly in the form of a user-space library (or system call) interface.

However, the new scheduler is being integrated into the AQuoSA open-source project[5] (Adaptive Quality of Service Architecture), which makes a well-designed user-space API and adaptive reservations available to application developers as a dynamically linkable library. AQuoSA provides a user-space library which implements the the AQuoSA API on top of the cgroup-based operations needed to deal with the IRMOS scheduler, easing the task of coding applications that want to use it.

4 More information is available at: http://libcg.sourceforge.net/.
5 More information is available at: http://aquosa.sourceforge.net/.

More details on the AQuoSA integration for the IRMOS scheduler can be found at: http://aquosa.sourceforge.net/news-2010-07-27.php.

RTLINUXFREE AND RTAI

RTLinuxFree is the evolution of RT-Linux, a modification (patch) to the Linux kernel aimed to support hard real-time applications, initially developed by Victor Yodaiken at University of New Mexico, then carried on by FSMLabs, which was acquired by WindRiver in 2007. It works as a small executive with a hard real-time scheduler that, in addition to the hard real-time tasks, executes the entire Linux OS (both the kernel and the user-space applications) as one of its low priority tasks [29].

RTLinuxFree adds a hard real-time executive between the hardware and the Linux kernel, which takes direct control of the interrupts of the hardware, passing them to the Linux OS only when they are not relevant for the hard real-time tasks. This way, hard real-time tasks run undisturbed and without interferences from the entire Linux OS.

Real Time Application Interface (RTAI) is a modification of the Linux kernel made by Prof. Paolo Mantegazza from Dipartimento di Ingegneria Aerospaziale at Politecnico di Milano (DIAPM). RTAI is an open-source project that builds on the original idea of RT-Linux, but has been considerably enhanced. RTAI allows to uniformly mix hard and soft real-time by symmetrically integrating the scheduling of RTAI proper kernel tasks, Linux kernel threads and user space tasks. However, similarly to RT-Linux, in RTAI Linux tasks run in the background with respect to the hard real-time kernel. Linux only executes when there are no real-time tasks to run, and the real-time kernel is inactive. Furthermore, the Linux OS can never block interrupts or prevent itself from being preempted by the real-time kernel.

Both approaches aim to integrate hard real-time tasks and standard Linux tasks within the same system. This allows, for example, running a set of real-time tasks that control an industrial plant together with the high-level software infrastructure and applications that allow monitoring of the plant from remote or from complex GUI-oriented interfaces. The main drawback of these approaches is that the real-time tasks cannot really exploit the full power of the hosting GPOS, and can only access a very limited set of peripherals (e.g. serial ports). A complex multimedia streaming application, that needs at least TCP/IP networking, heavy use of the disk and access to audio and video adapters would not be suitable to run on such a system.

B) *Exception-based Management of Time Constraints Violations*

As shortly seen above, Operating Systems are being continuously enriched with more and more features for handier development of time-sensitive, (mainly soft) real-time and multimedia applications. This allows the development of applications with stringent timing requirements, provided the programmer are also given some mechanism for specifying these timing constraints within the application, and for dealing with their (hopefully occasional) violations.

In fact, in many cases, good resources saturation level is needed as much as meeting the timing requirements is. Moreover, timing requirements may be stringent, but in most of the cases not safe-critical, and it may be sufficient to fulfil them with a high enough probability.

Therefore, timing constraints violations should be expected to occur at run-time, and developers must somehow cope with them. Specific real-time oriented programming language and compiler extensions can be of great help, for example by allowing dealing with timing constraints specification and violation with an exception-based management programming paradigm, similarly to what already exists for other kind of exceptional events in in languages like C++, Java [8] or Ada [9]. In these contexts, an exception is raised during program execution either because of programmer's explicit intentions, i.e., it is thrown using a function usually called throw, or implicitly, because of some illegal operation, such as non existing file, forbidden access to memory, etc. Moreover, developers are generally asked to specify which are the code segments that may be affected by this phenomenon by enclosing them in a special block (e.g., `try { ... } catch`).

More specifically, an effective real-time enabled language/compiler, should allow specifying at least two forms of timing constraints: deadline constraints, i.e., a software component needs to complete within a certain (wall-clock) time, and WCET constraints, i.e., a software component needs to exhibit an execution time that is bounded.

Of course, such mechanisms also require some support not only at the programming language and/or compiler level, but also in the underlying platform and OS. For example, a fully POSIX.1b [10] compliant Operating System, already exhibits a set of real-time extensions that may be sufficient to the enforcement of real-time constraints, as well as to the development of software components exhibiting a predictable timing behavior. However, working directly with these very basic building blocks is definitely non-trivial. The code for handling timing constraints violations, as well as other types of error conditions, needs to be intermixed with regular application code, making the development and maintenance of the code overly complex. Including an higher level support in the programming language would, instead, improve usability of these building blocks, by enabling the adoption of an exception-based management of these conditions.

POSSIBLE UTILIZATION SCENARIOS

In order to derive the requirements for such kind of mechanism two typical use cases are considered: (i) a component based multimedia player, and (ii) an anytime algorithm implementation coming from a control scenario. Code examples are given in a language resembling the syntax of the the C programming language.

For (i), consider a multimedia player, designed as a single thread of execution activated periodically or sporadically. A possible behaviour for the Video Decoder component of such application is outlined in the UML Activity Diagram of Figure 10.

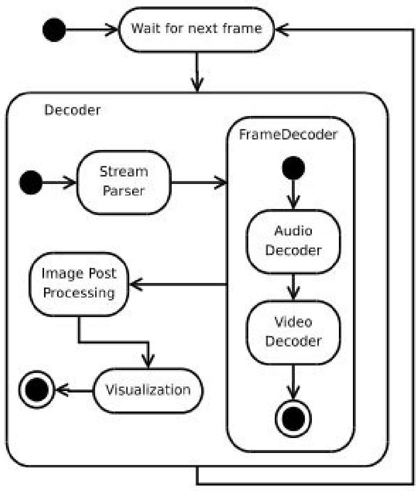


Figure 10: UML Activity Diagram for a possible video decoder.

From a run-time perspective, both audio samples and video frames must be decoded and played within precise timing, depending on the type of the media and on the format of the stream. It is well-known that, if data are not ready on time, it may be better to abort the operation on the current frame and start working on the next frame, since the user perception would not benefit from the reproduction of late samples. Similar considerations can be made for the frame post-processing part, which might be skipped if the decoder is lacking time. Therefore, the player described by Figure 10 may be implemented as follows:

```
void FrameDecoder(raw_frame_t f, dec_frame_t *d) {
AudioDecoder(f, d);
VideoDecoder(f, d);
}

void VideoDecoder(stream_t *s) {
static dec_frame_t d;
raw_frame_t f;

while (1) {
f = StreamParser(s);
try (T) {
  /* Aborted if still inside at time T */
    FrameDecoder(f, &d);
    ImagePostProcessing(&d);
  }
  /* If aborted, re-use last decoded frame */
Visualization(d);
wait_for_next_frame(s);
}
}
```

On the other hand, from a designer perspective, it would be highly desirable to characterize each component with a WCET (or with some other appropriate statistic of execution time distribution). Also, it might be desirable that Video Decoder actually respects such WCET, even in cases of overload, e.g., when a frame is particularly difficult to decode. Due to the in-place timing requirements, it would be useful to characterize the Frame Decoder invocations that happen inside the Video Decoder with the WCET to be expected at run-time as well, since the sum of such value, plus the WCETs of the Stream Parser, Filtering and Visualization components, turns out to be WCET of the Video Decoder itself. Moreover, video decoding architectures are highly modular, and make heavy use of third-party video and audio decoding plug-ins, e.g., depending on the stream format. Thus, in order to allow for an appropriate use of Frame Decoder within real-time applications, it would be highly desirable for libraries developers to have some mechanism for specifying a WCET estimation such that either: (1) the decoding operation terminates within the WCET limit, or (2) it is aborted.

For what concerns (ii), anytime algorithms, they have been theorized in real-time systems from long time, for enhancing flexibility [11][12]. Thus, whenever it is possible, the computation done at each activation is split in a mandatory part, that always needs to be completed, and one or more optional parts, that may be executed if there is enough time. There are a lot of examples in graphics, multimedia and image processing that might perfectly fit this definition. They are also utilized in embedded control, as in Quagli et al. [13], where such paradigm is applied for controlling the stability of an helicopter simulator. In fact, if an accurate enough estimation of the duration of the mandatory and of all the optional computation phases is available, and if a facility like `rmng_computation_time()`, capable of reading the time left for the current instance of a `try{ … } catch` block, is provided, then an anytime algorithm may be coded just checking, before entering each optional computation, if there will be enough time to complete it. However, if the optional parts exhibit fluctuations in their actual execution time, relying on conservative estimations for that may result in dropping them more often than when strictly required. Thus, an alternative solution is to always attempt to execute the entire computation, having the optional parts asynchronously interrupted by an exception if they are lasting longer than allowed, as it is shown below:

```
int D1 , D2 ;   /* Computation times of the optional parts
*/
int res;

while (1) {
start_computation();
res = mandatory_computation();
try (D1 + D2) { /* Aborted if execution exceeds */
   res = optional_computation_1(res);
   res = optional_computation_2(res);
} catch {
   end_computation(res);
}
wait_next_period();
}
```

Notice how the overall results comes from the “merge” of the intermediate results of the various computation phase, performed by each computation phase itself (this is why the result of the i-th phase is passed as argument to the (i + 1)-th one), and actually utilized only at the time of the `end_computation()` call. In fact, should any optional part be aborted by an asynchronous exception, the result of the computation should either completely include the last optional computation results or completely ignore them.

Finally, it must be noted that there are cases where it may not be possible to asynchronously jump from an arbitrary position in the application code, to the exception handling logic. This implies the application should be designed so as to tolerate this kind of operation abortion, avoid possible memory leaks, and to properly cleanup any resources that might be associated with the aborting code segment. To adequately support this, it would be useful to have, in this set of programming language extensions a mechanism to temporarily stop the notification of an exception and the jump to the recovery logic for a group of statements. Obviously, the notification should reach the application anyway, and remain pending till the end of the “protected” code section. This way, the proposed approach can be used for detecting violation of a timing constraint even in these cases, and, moreover, the recovery logic can, in such cases, rely on the application data to be consistent, since the computation was not interrupted asynchronously. Obviously, should the application need to compensate for the accumulated delay, then it would be desirable to have a means provided allowing to retrieve how much such delay is.

REQUIREMENT DEFINITION

From the above considerations, the following set of high-level requirements may be identified for the envisioned mechanism:

- Mandatory requirements:
    a) it should be possible to associate a deadline constraint to a code segment, either specifying relative or absolute time values;
    b) it should be possible to associate a WCET constraint to a code segment;
    c) when a timing constraint is violated, it should be possible to activate appropriate recovery logic that allows for a gracefully abort of the monitored code segment; also, it should be possible for the recovery code to either be associated to a generic timing

constraint violation, or more specifically to a particular type of violation (deadline or WCET);

d) it should be possible to use the mechanism at the same time in multiple applications, as well as in multiple threads of the same application;

e) nesting of timing constraints should be allowed, at least up to a certain (configurable) nesting level;

f) if there are two (or more) nested timing constraints, a violation should be propagated in such a way that it is caught by the recovery logic associated with the code segment that caused it to occur;

g) it should be possible to cancel a timing constraint violation enforcement if the program flow runs out of the boundary of the associated code segment, e.g., when it ends normally or when another kind of exception requests abort of the code segment;

h) the latency between the occurrence of the timing constraint violation and the activation of the application recovery code should be known to the designer/developer, and it should be possibly negligible with respect to the task execution time;

i) the mechanism should allow the programmer to specify some “protected” section of a code segment that will never be interrupted by a timing constraint violation notification. Thus, if that happens, the execution of recovery code would be delayed while inside such a section.

j) the mechanism should provide a method for monitoring the time remaining before the specified constraint violation to occur;

- Optional requirements:

  a) the mechanism could provide a method for checking if a constraint violation has been notified with the correct timing and, if not, how much is the difference between the expected and the actual (i.e., the latency) notification of such violation;

  b) the mechanism could provide support for gathering benchmarking data of the code segments, instead of enforcing their timing-constraints. This operational mode could be enabled at compile time, and used for tuning the actual parameters used as timing constraints for the various code segments.

PROPOSED APPROACH

Here a mechanism complying with the requirements identified above is presented, with a focus on the programming paradigm and syntax, by means of the following constructs:

- `try_within_abs (try_within_rel)`: starts a try block with an absolute (relative) deadline constraint;
- `try_within_wcet`: starts try block with a maximum allowed execution time;
- `try_within_disable` and `try_within_enable`: suspend and re-enable, respectively, the notification of a timing exception. Notifications that reach the application after a disable are not lost, rather they are deferred until the next enable;
- `ex_timing_constraint_violation`: is the basic type for timing constraint violation exceptions; catching this will actually intercept any kind of timing constraint violation, without distinguishing between them;

- ex_deadline_violation: is what occurs when a try_within_rel (or try_within_abs) segment does not terminate within the specified time;
- ex_wcet_violation: is what occurs when a try within WCET segment executes more than the specified time.

Below it is shown, again, how the decoder imagined in Figure 10 can be implemented with the help of these extensions. Let us assume that an estimation of the decoding time is known to be 12ms, and that the presentation time (pts) of the next frame extracted from the stream can be used as the deadline for the decoding and the visualization of such frame.

```
int FrameDecoder(raw_frame_t f, dec_frame_t *d) {
   int rv = 0;

   try_within_wcet (12000) {
      AudioDecoder(f, d);
      VideoDecoder(f, d);
   }
   handle
      when (ex_wcet_violation) {
         rv = -1;
      }
   end;
   return rv;
}

void Decoder(stream_t *s) {
   raw_frame_t f;
   dec_frame_t d, d_old ;

   while (1) {
      f = StreamParser(s);
      try_within_rel(f->next_frame_pts) {
         if (FrameDecoder(f, &d) == 0)
         ImagePostProcessing(&d);
      }
      handle
         when (ex_deadline_violation) {
            d = d_old ;
         }
      end;
      Visualization(d);
      d_old = d ;
      wait_for_next_frame(s);
   }
}
```

As a final remark, the example code below shows a typical usage of the disable/enable mechanism to protect code segments that must not be interrupted asynchronously.

```
struct my_object *p_obj = NULL;
try_within_abs(next_dl) {
   /* safely interruptible computations */
   ...
   try_within_disable();
   p_obj = malloc(sizeof(struct my_object));
   if (p_obj == NULL)
      throw(EnoMemoryException);
   my_object_init(p_obj);             /* Constructor */
   try_within_enable();
   /* safely interruptible computations again */
   ...
} finally {
   /* Free allocated resources */
   if (p_obj != NULL) {
      my_object_cleanup(p_obj);   /* Destructor */
      free (p_obj);
      p_obj = NULL;
   }
} handle
   when (oml_ex_deadline_exception) {
      /* Recovery logic */
      …
   }
end;
```

Both the memory allocation for the new object and its construction (which in turn may involve further allocation of memory segments, and/or other OS resources) are made atomic with respect to deadline exceptions. Also, destruction of the object occurs in the finally statement, what ensures it always happens, even if an exception is raised within the application body, which is not caught by the when clause immediately following.

PROOF OF CONCEPT IMPLEMENTATION

As a proof of concepts, the described mechanism has been implemented (for all POSIX systems) as a shared library for the C programming language, comprising a set of C preprocessor macros realizing the illustrated semantic. For detailed analysis the interested reader can look at [14] and [15]. In these works, the usability and the performances of such implementation have been evaluated by modifying a commonly available multimedia player (`mplayer` [16]) for using it. In the experiments, the number of dropped frames, the maximum and average Audio-Video delay and the maximum inter-frame time (IFT) exhibited by the original and modified version of the player, while displaying an H.264 movie trailer ("Big Buck Bunny", [17]) have been measured for 100 independent runs, and under no, light and heavy system load conditions. What would have been desirable for a smooth and regular playback was the number of dropped frames and the A/V delay to be as small as possible (ideally zero), and the IFT, given the video is 25fps, to stay around 40ms.

In the set of graphs in Figure 11, the curves representing the original (`mplayer`) and exception-modified (`mplayer-dlex`) versions of the player are very similar, if not super-imposed, meaning that the behavior of the twos almost perfectly matches, and thus that introducing the exception-based handling of frame dropping does not entail misbehaviors. The second set of results follows the same scheme, with the following differences:

- "*ORIG and DLEX Light CPU Load*" refers to runs of `mplayer` and `mplayer-dlex` disturbed by the stress ([18]) benchmarking program running 1 CPU and memory intensive thread, plus 1 HDD massive reader/writer thread;
- "*ORIG and DLEX Heavy CPU Load*" refers to runs of `mplayer` and `mplayer-dlex` disturbed by 2 CPU and memory intensive threads and 1 HDD massive reader/writer thread.

Figure 11 shows that using exception-based frame dropping does not make the number of dropped frames much different, in fact there is no clear domination of one curve on the others, spanning all the runs in neither of the two load scenario. However, the maximum IFT experiences fewer and smaller peaks in the DLEX cases and the A/V delay curves for DLEX are almost always below the ORIG ones. This means that `mplayer-dlex` is able to achieve more regular behaviour than the original `mplayer`. Therefore, it is possible to conclude that the proposed programming level extensions, since they enable capturing the specific timing behaviour of time-sensitive applications, would bring beneficial effects for them and their developers.

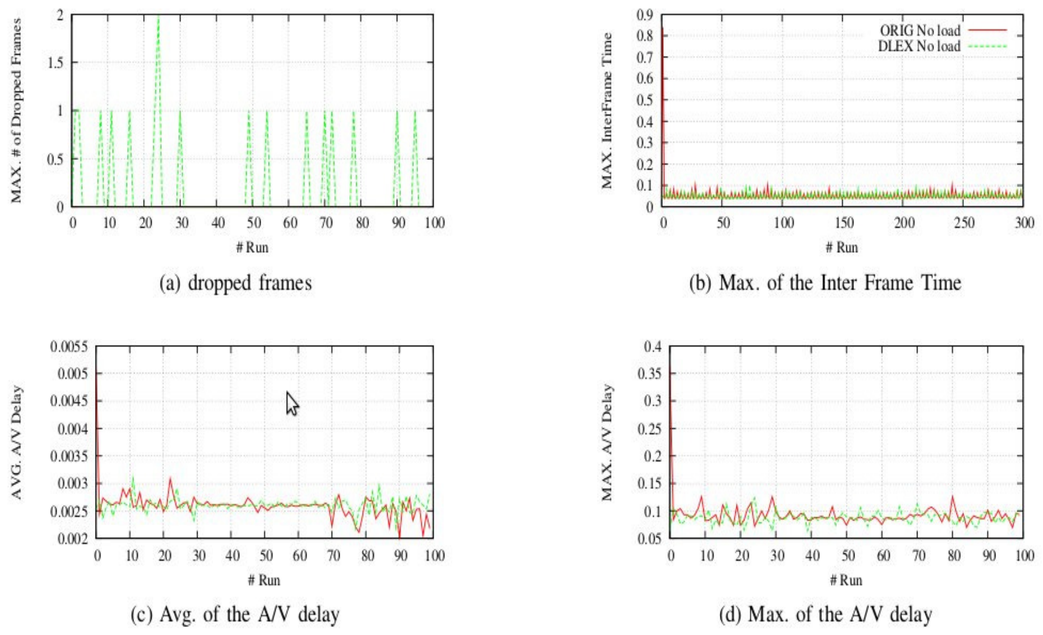


Figure 11: Results for the 100 runs of mplayer (ORIG curves) and mplayer-dlex (DLEX curves) without load: (a) total number of dropped frame; (b) maximum inter-frame time; (c) average A/V delay; (d) maximum A/V delay.

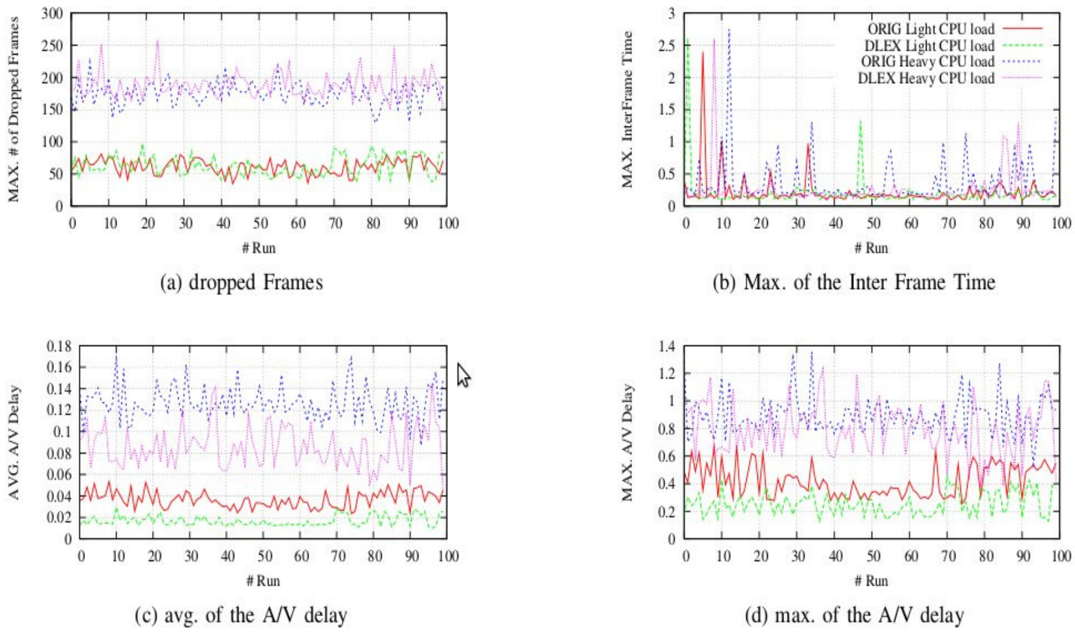


Figure 12: As above, but with some system load.

C) Distributed Deadline Synchronization Protocol

Many distributed and multiprocessor real-time applications consist of pipelines of tasks that must complete before their end-to-end deadlines. For example, this is the case for the important class of multimedia applications targeted in S(o)OS together with other application classes.

For this type of applications, it is recurrent to use at run-time scheduling policies based on Fixed Priority (FP) or Earliest Deadline First (EDF), and an accurate analysis phase is needed in order to assess the schedulability of the applications, i.e., to verify their capability to actually meet their deadlines once deployed and launched.

Different schedulability analyses have been proposed for both FP and EDF scheduling. All the schedulability analyses proposed so far assume that a global clock synchronization protocol is used to synchronize the deadlines of jobs allocated on different processors. This assumption may limit the applicability of EDF to such systems.

In S(o)OS, we propose the Distributed Deadline Synchronization Protocol (DDSP) for computing the absolute deadlines of jobs. DDSP does not require a global clock synchronization, yet existing schedulability analyses are valid for schedules generated by DDSP.

For the sake of brevity, DDSP is not described in this document and the interested reader is reminded to the published paper [40] for details.

D) Hierarchical Real-Time scheduling on multiprocessors

In S(o)OS, we developed a theoretical framework for designing hierarchical scheduling systems that covers all phases of the design :

- a novel interface model, called bounded-delay multipartition interface ;
- a schedulability test for an application that can be used to derive interface parameters ;
- an allocation policy, called fluid bestfit .

This framework can be applied, for example, to the IRMOS hierarchical real-time scheduler described above, for performing a precise schedulability analysis of a set of real-time applications deployed on a multi-processor/multi-core platform.

Again, for the sake of brevity, the details of this technique are omitted and the interested reader can refer to the published paper [41].

IV.Speculative Execution

1. Transactional Memory

Transactional memory (TM) is a very promising technique for concurrent programming that has attracted a lot of attention recently. In this section, we focus on applying transactional memory to distributed systems. In particular, we focus on machines with processors *without* cache-coherence, which is a novel target for transactional memory. We also discuss how to predict the scalability and performance of applications that use transactional memory to understand better whether the particular application can indeed benefit from the use of transactional memory or not.

A) Distributed Transactional Memory

As multicore architecture is becoming the norm in manufactured computers, simple concurrent programming abstractions are necessary to help application programmers leverage their power. Since Intel COO announced the multicore revolution in Intel product line-up in 2004 at San Francisco, other manufacturers have enforced a similar change in their production. Sun Microsystems have proposed its 8-core UltraSPARC and UltraSPARC 2 CPUs in 2005 and 2007, respectively, AMD presented its 2-core and 4-core Opterons CPU between 2006 and 2007, and in 2010 Intel proposed its 6-core Westmere as an improvement over its well-known Xeon Servers. Hence, the tendency is no longer to increase the frequency of a single processor but to increase the number of cores on the same CPU that concurrent programming can exploit.

Transactional memory (TM) is a concurrent programming abstraction simple to reason with. This abstraction operated a complexity shift from the implementation of concurrent applications to the implementation of the transactional memory itself as it acts as a black-box that hides all complex synchronization mechanisms such as locks, compare-and-swap, etc.

The transactional memory transparently manages the accesses to the shared memory and may decide to abort and restart the whole transaction in case of conflict with another concurrent transaction. The transactional memory guarantees that the series of encapsulated actions appears as if it was executed atomically (if the transaction commits) or not at all (if it aborts).

New research challenges arise when applying transactional memory to message passing systems. These challenges include:

- the definition of a cluster-based distributed transactional memory where objects are replicated at all nodes of the systems;
- the combination of the memory transaction with the distributed transaction in a unified execution starting from the client request to the shared memory accesses of each replicated servers; and
- the potential definition of a directory protocol as a building block of a distributed transactional memory for large-scale networks and in which the object is locally acquired by a node prior to being updated.

We have started investigating DTM on non-replicated systems to fit the needs of existing non-cache-coherent architectures. To this end, our algorithm builds upon the RCCE message-passing library and the Intel Single-Chip Cloud Computer (SCC) architecture presented in *Deliverable 3.2.*

ALGORITHM OVERVIEW

The proposed algorithm aims to create a *fully decentralized*, modular, and scalable} DTM system that guarantees *opacity* (i.e., atomicity of transactions) and *lock-freedom* (should be provided by the contention manager used). The algorithm, as described below, operates on a global address space model (the nodes have access to a shared memory), but can be adjusted to use a PGAS memory model. The most important characteristics are:

1. **Distributed Locking Service.** The algorithm implements a Distributed single-writer/multiple-readers locking protocol that is used to protect the accesses to the shared memory. The protocol distributes the metadata among the system's nodes.
2. ***Contention Manager.*** A Contention Manager is responsible for selecting the appropriate action when a conflict is detected. A different policy can potentially be applied for each of the three conflict cases (read/write, write/write, write/read).
3. **Eager read acquisition/Visible reads.** Transactional reads are visible to the other transactions.
4. **Lazy write acquisition/Deferred-writes.** The update events are buffered locally until the commit phase. Only during the commit phase the transaction tries to acquire the write-locks and write to the shared memory.

The algorithm consists of three distinct functional parts (*object locating*, *distribute-locking*, and *contention-management*) so different design choices and extensions can be easily adjusted. For example, data replication can be achieved with an additional network step by configuring the distributed locking service to publish the writes.

PSEUDOCODE

We present our DTM algorithm by presenting the API it offers to write highly concurrent programs. The programmer has simply to delimit region of codes the execution of which should look as atomic from the standpoint of other concurrent transactions. The shared memory accesses within this region are simply wrapped by the dedicated DTM algorithm to provide this atomicity guarantee.

Transaction start. Starts a new transaction.

```
txstart() {
      create_metadata();
    }
```

Transactional read. Node i reads a shared object obj from the memory within a transaction.

```
txread(i, obj) {
  //either in write or read buffer
  if((obj_buffered = get_buffered(obj)) != NULL) {
    return obj;
  }

  nId = get_responsible_node(obj);
  tx_metadata = get_metadata();
  /*similar to an RPC-like call on node nId*/
  resp = rlock(nId, i, tx_metadata, obj);

  if (resp == true) {
    value = shmem_read(obj);
    add_read_buffer(obj, value);
    return value;
```

```
   } //contention manager aborted current tx
   else {
      txabort();
   }
}
```

Transactional write. Writes value val on a shared object obj within a transaction.

```
txwrite(obj, val) {
   update_write_buffer(obj, val);
}
```

Transaction commit. Node *i* commits a transaction.

```
txcommit(i) {
   //write_buffer: the non-locked writes
   //writes_locked: the locked writes
   while((item = getitem(write_buffer)) != NULL) {
      nId = get_responsible_node(item.obj);
      tx_metadata = get_metadata();
      /*similar to an RPC-like call on node nId*/
      resp = wlock(nId, i, tx_metadata, item.obj);

      //contention manager aborted current tx
      if (response == false) {
         txabort();
      }

      append(item, writes_locked);
   }

   while ((item = getitem(writes_locked)) != NULL) {
      shmem_write(item.obj, item.val);
   }

   //iteratively call wlock_release()
   wlock_release_all(i, writes_locked);
   //iteratively call rlock_release()
   rlock_release_all(i, read_buffer);
}
```

Transaction abort. Node *i* aborts a transaction.

```
txabort(i) {
   /*release all read and write locks*/
   wlock_release_all(i, writes_locked);
   rlock_release_all(i, read_buffer);
}
```

Read-lock Release. Node *i* releases lock for object *obj*.

```
read_lock_release(i, obj) {
   remove_reader(obj, i);
}
```

Write-lock. Node *i* acquires write-lock for object *obj.*

```
write_lock(i, tx_metadata, obj) {
   enemy_tx = write_after_write(i, obj);

   if (enemy_tx != NULL) {
      c1 = contention_manage(tx_metadata, enemy_tx, WAW);
   }

   //contention manager aborted enemy
   if (c1 == true) {
      //multiple readers may exist for obj
      enemy_tx_list = write_after_read(i, obj);

      if (!is_empty(enemy_tx_list)) {
         c2 = contention_manage(tx_metadata,
               enemy_tx_list, WAR);
      }

      //contention manager aborted enemies
      if (c2 == true) {
         set_writer(obj, i);
         return true;
      }
   }
}
```

Write-lock release. Node *i* releases lock for object *obj*.

```
write_lock_release(i, obj) {
   set_writer(obj, NULL);
}
```

B) Predicting Scalability of Transactional Memory Applications

Conducting a thorough performance evaluation of an STM is very time consuming. Depressingly, even with all this effort, and even with the same application, it can still be hard to predict the performance if the number of underlying threads on which the application needs to be deployed is different than those of the experiment. Basically, one might have to conduct an entire set of new experiments to get some understanding of the performance of the STM with the new number of threads.

We propose a pragmatic approach to contribute to changing this state of affairs. Using classical engineering approximation techniques, we extract from a set of STM performance measurements, analytical performance functions to model the scalability of the STM. We show, more specifically, that polynomial and rational functions provide good interpolations of STM performance: even with only a handful of measurements, the average error in most cases is around 1-2%. Further, we show that we can perform reasonably precise extrapolation using rational functions: basically, using measurements with up to m threads, we can predict the performance up to roughly 2m threads with a relatively low error (around 10% in best cases).

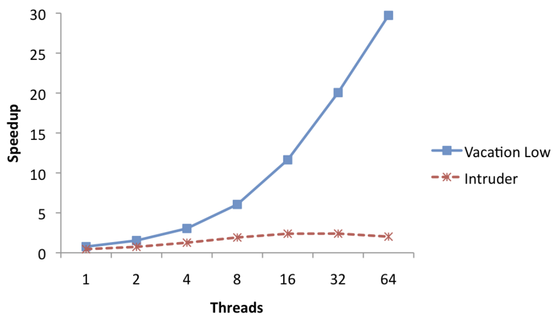


Figure 13: STM speedup on two STAMP benchmarks

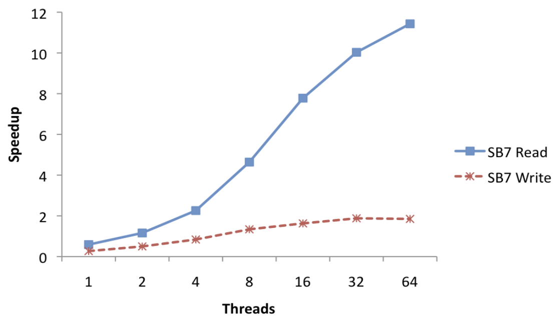


Figure 14: STM speedup on two STMBench7 workloads

The figures above depict the speedup of a parallel code that uses SwissTM [1] over sequential, non-instrumented code, for two different workloads from the STAMP benchmark suite [2]. The figure conveys the very fact that, while STM scales very well on the *vacation low* workload, outperforming sequential code by almost 30 times with 64 threads, its scalability is not nearly as good on *intruder*, where it outperforms sequential code by only 2 times with 64 threads. In fact, even the same application that uses STM can have highly varying performance depending on the workload configuration. The lower contention *read-dominated* STMBench7 [3] workload achieves a speedup factor of almost 12 with 64 threads, but the higher contention *write-dominated* workload of the same benchmark is faster than non-instrumented sequential code by less than 2 times.

In fact, two recent studies [4][5] drew contradictory conclusions about the scalability of an STM even on the same subset of benchmarks. Not only the performance of different STM workloads can differ significantly, but it is also very difficult to predict how it will evolve should the number of available threads (cores) be increased. In general, experience shows that even if the scalability of an STM looks great for a range of thread values, performance might actually (slightly or significantly) drop after some point: basically, contention can simply become too high with too many threads. But knowing at which point this happens is hard without extensive experiments. For the workloads in Figures 2 and 13, the performance peaks at 22 threads for *intruder*, 32 for *STMBench7 write*, 52 for *STMBench7 read*, while for *vacation low* it keeps improving up to 64 threads.

Clearly, this state of affairs might put potential adopters of STM in a difficult position: should they write their application with an STM in mind and expect the performance to speed up with a new, to be purchased, architecture with more cores? Or should they simply keep hacking their old lock-based techniques and forget about the STM?

Certain rules of thumb do exist. For example, if different application threads mostly access disjoint data, and if the majority of accesses are reads, resulting in lower contention, the application is probably a good candidate for parallelization with STM and scaling it to a larger number of threads would certainly reveal beneficial. Also, if only a small portion of the code is actually using transactions, the overheads of STM will not be significant and the application is likely to benefit from STM. However, these rules are somewhat vague and not easily applicable in all cases.

We explore a simple and pragmatic approach to predicting the scalability of STM-based applications. We employ classical engineering approximation techniques to predict the performance of a workload *W* with *n* threads, based on its performance with m_i $(1 \le i \le M)$ threads. Put differently, we construct an analytical *performance function performance = f(n)*, based on *M* performance measurements. Function *f* describes the characteristics of the application workload and the whole computing environment (STM, OS, hardware etc.). It takes as a parameter the number of threads n, and outputs a guess *f(n)* of the expected performance. Interestingly, the approach does not require any knowledge of the workload or the system configuration when constructing *f*: no access to the source code of the application is required.

Basically, our approach to predicting performance consists of three steps:

1. **Profiling.** We measure the performance of the workload on a target STM and computer system with several thread counts.
2. **Approximation.** Using classical approximation techniques, and based on the performance measures collected during the profiling step, we construct a performance function *performance = f (n)* supposed to capture all the characteristics of the workload, STM and the computer system.
3. **Prediction.** Finally, we use *f* to predict the performance with a different number of concurrent threads than we used during the profiling step.

INTERPOLATION

We discuss and evaluate below various possible function choices for interpolating STM performance. The choice of the approximation function type is very important and can impact the error of the approximation significantly. The ideal function is one that is relatively simple, yet general, and can be applied to different systems and workloads. We considered several standard choices:

1. Polynomial functions.
2. Rational functions.
3. Logarithmic functions.
4. Exponential functions.
5. Rational functions.

Polynomials can be used to approximate any continuous function on a closed interval to any degree of accuracy. Rational functions have the same nice property, but they can model more diverse behaviours than polynomials. The major drawback of polynomials and rational functions is their not conveying much information about the workload behaviour. On the other hand, logarithmic, exponential and power functions convey more information about the workload performance, and they could be used to approximate some of the workloads very well. Unfortunately, as the experiments below reveal, they could not be used for all the workloads equally well. Our experiments led us to use polynomial and rational functions as they, maybe unsurprisingly, proved to be the most general (Figure 15).

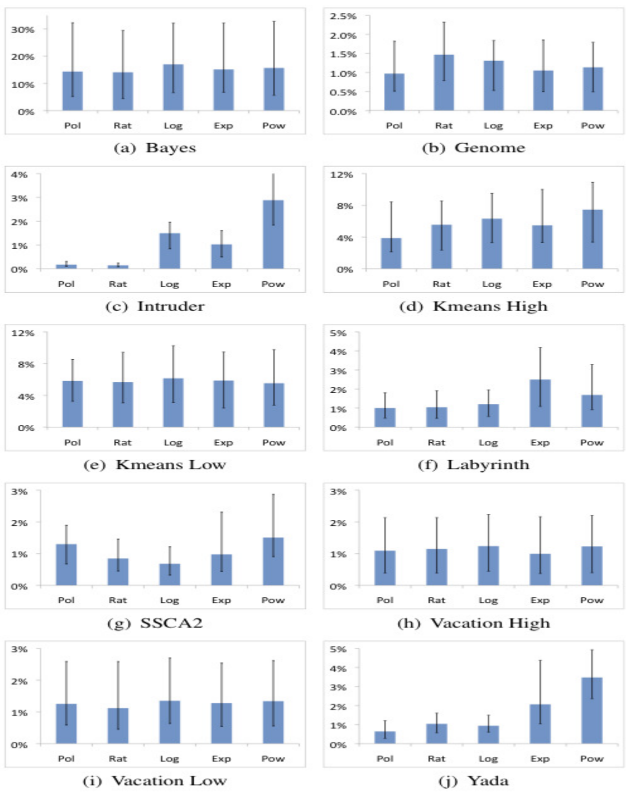


Figure 15: Approximation errors for STAMP

Clearly, polynomial and rational functions revealed to be the best suited for approximating STM performance, both because of the low approximation error, and uniform and simple approximation functions.

EXTRAPOLATION

We describe and evaluate here the use of performance functions to extrapolate performance, i.e. predict performance for more than *m* threads, based on several measurement all with less than *m* threads. This is, arguably, the most interesting usage of performance functions.

We constructed rational and polynomial functions based on the measurements with 1 to 8, 1 to 16 and 1 to 32 threads. Figure 16 conveys the error of extrapolations for these cases.

The figure shows that the polynomial functions are not very useful for extrapolation (which might not be very surprising) as the error increases rapidly after 32 threads.

The figure also shows that the extrapolation of STM performance data can be performed using rational functions for many workloads.

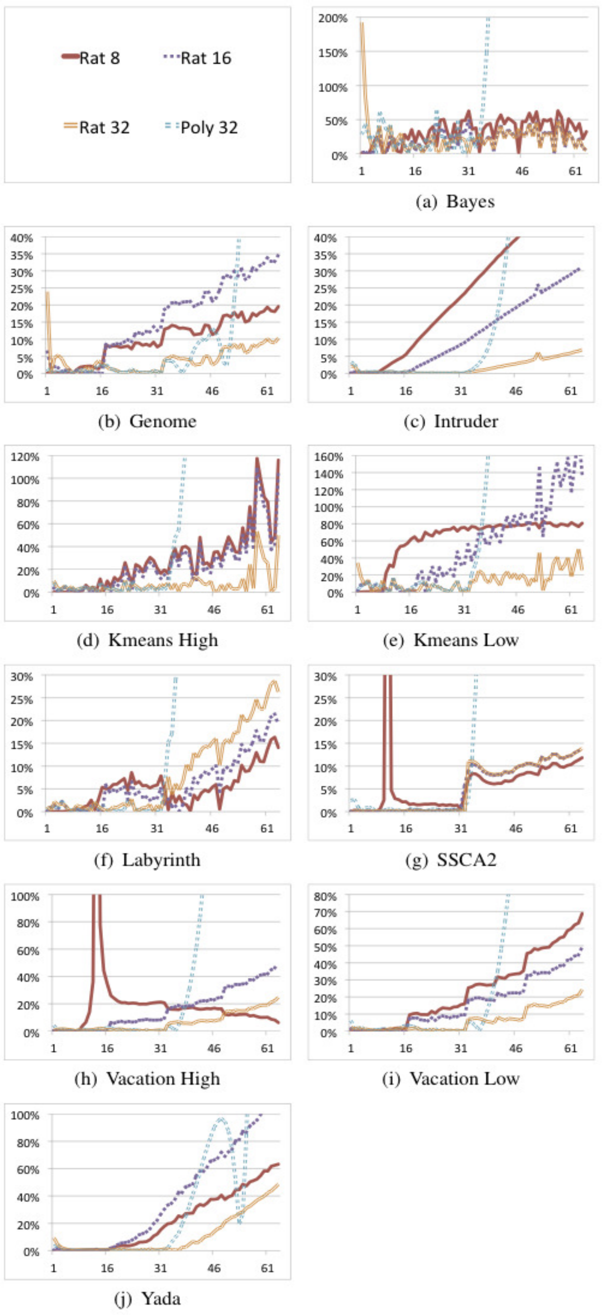


Figure 16: Extrapolation errors for STAMP

DISCUSSION

We do not advertise our approach as the silver bullet to help make the right choice for the use or not of an STM in a given application and workload. The most important limitation of our work is that the performance function *f* strongly depends on both the workload and the computer system. Small changes in the workload or the system might introduce significant errors in the predictions. For example, even replacing two 4-core CPUs with a single 8-core CPU, or increasing the frequency at which a certain lexical transaction is executed might significantly change the performance function.

We believe that our approach could also be useful for other types of concurrency schemes, e.g. lock-based techniques. It might not be as promising as with STM, because lock-based code is typically more difficult to write than the code that uses STM and, thus, the programs required for profiling runs are more difficult to obtain. Furthermore, overheads of STM are typically much higher, making the question of performance prediction more important. Also, we believe that our technique can be, rather straightforwardly, applied to upcoming Hardware and Hybrid Transactional Memory and that it will, most likely, be relevant for both.

In the future, we plan to improve approximations using the knowledge about the modelled system. For example, we know that the performance cannot be negative, and also, that it typically changes rather slowly. This could help us remove some local minimums and maximums in the predicted function. Also, using runtime information provided by the STM and the computer system could help us improve our approximations. In particular, it would be very interesting to focus on predicting the performance for machines with low number of hardware threads (desktop machines). In these cases, not many measurements can be collected and used for function fitting and the runtime information could be used instead to achieve better approximations.

2. Real-Time Transactional Memory

Both real-time computing and transactional memory have something to say about concurrency management, although their approach to that is quite different. In fact, real-time scheduling analysis often relies on very pessimistic estimations of computation times and resource sharing patterns, in order to be able to derive guarantees that can be considered valid even in the worst possible scenario. This involves protecting each critical section with mutually exclusive locks and taking into account both, the critical section durations and the blocking times threads will incur on, while analysing the system. On the other hand, transactional memory optimistically allows more than one thread inside critical sections, intervening only upon real conflicts of multiple threads on the same memory locations. In fact, this often allows one to achieve a much higher throughput, but can completely defeat any expectation of predictability and determinism.

However, in the domain of the so called soft real-time systems, full off-line knowledge of all the system and applications parameters and characteristics is very unlikely and, at the same time, dimensioning around worst case estimations for them would lead to great amount of resources underutilization. Therefore, in these cases, borrowing some optimistic thinking from the transactional memory approach, for what concerns accessing data shared between different threads, could bring interesting benefits.

The key idea is to envision a (some) mechanism(s) for enabling the typical transaction-based access to critical code sections, thus obtaining increased parallelism and resources saturation, while still retaining the strict prioritization and temporal isolation (depending on which specific scheduling algorithm is considered) that characterize real-time systems, and that allows the applications running on them to expect deterministic service levels.

A) Existing Work

Some work toward the same or similar objectives to the ones expressed above has already been done in the last years. This section will briefly go through it and explain why there is still wide room for improvements.

Without any doubts, one of the most comprehensive work on transactional memory is what is known under the name of TxLinux [42], i.e., a modification of the Linux kernel to use transactions instead of locks for many of the critical sections it contains. Even more interestingly, a mechanism very similar to the real-time resource sharing protocol known as Priority Inheritance [43] is utilized in order of avoiding the intermixing of locks and transactions to lead to a priority inversion situation for critical real-time threads. However, up to now, TxLinux only consider hardware transactional memory, meaning that, due to the lack of actual chips implementing such support directly on the silicon, it is still something which is “confined” to some very specific architectural simulator.

In [44] Maldonado et al. tries to intermix scheduling and (software) transactions on Linux. They achieve a good compromise between throughput and timeliness, but still with a mostly heuristic approach, rather than by exploiting the scheduling specific information of the involved threads. For example, the amount of time-slice extension guaranteed to threads running transactions is experimentally determined, and this might compromise the guarantees that any real-time scheduling algorithm strives to provide (and the same holds for the fair distribution of CPU time a typical general purpose scheduling algorithm aims at). For instance, instead of just enlarging it up to a certain maximum value N, what can be done is to have the conflicting threads donating their own time-slice to the thread whose transaction continues being executed (starting from the very instant when the conflict happens).

In [45] the same authors consider also transactions with deadlines, which is a more typical real-time constraint, and they also affect the Linux scheduler even farther, as soon a transaction with a deadline starts to execute. However, the deadlines are assigned to transaction only, and are not actually used to schedule the threads, but just as a countdown handled by the contention manager in order to decide – depending on how close the deadline for an aborting transaction is – in what “mode” it should be restarted, i.e., if it can suffer for more aborts or should complete uninterrupted to not miss it. Moreover, the thresholds for switching between the different transaction execution modes is still heuristically determined.

In [46] Schoeberl et al. Introduce some analysis techniques for meeting real-time constraints while using transactional memory. However, the focus is still just on hardware transactional memory, and just for multicore architectures. Moreover, the analysis poses its foundation on very limiting simplifying assumptions, like only 1 critical section per each thread with the same length for all the threads.

Finally, in [47], Sarni et al. propose an interesting modification of an existing software transactional memory library, but they fail to explain how deadlines are assigned to transactions and, more important, how this affect the contention-management logic.

Summarizing, in almost all the existing works about soft transactional memory (STM), it is not clear, upon conflict, which transaction is stopped. There is some support for dealing with priorities during contention management, and methods for the transaction management to interact with the scheduler have been proposed, but a great deal of heuristic is still what can be found in all the described solutions. This makes perfectly sense, from the STM point of view, since the main purpose there is to always be able to make the choice that will maximize the overall throughput. However, from a real-time perspective, applying some sort of simple yet strict policying here seems quite important, for enabling higher levels of determinism and predictability. For instance, the thread with lesser priority could be the one that is asked to abort, and/or the priority of the thread which continues could be boosted up to the maximum priority among all the ones that it is in conflict with.

B) PRELIMINARY EXPERIMENTS

Some preliminary experiments have been performed by modifying Elastic STM [51] running on GNU/Linux. More specifically, the logic for changing the scheduling policy and/or priority of a thread that enters a transaction has been introduced. The hardware platform on which tests have been run is a Dell PowerEdge R815 server equipped with 64GB of RAM, and 4 AMD Opteron 6168 12-core processors (running at 1.9 GHz), for a total of 48 cores. From the point of view of the memory hierarchy, each processor has two 6-core NUMA nodes and is attached to two memory controllers. The memory is globally shared among all the cores, and the cache hierarchy is structured on 3 levels: private per-core 64 KB L1D and 512 KB L2 caches, and a global 10240 KB L3 . Cache lines are 64B long. The utilised benchmark is part of benchmark suite used in [51].

The main idea was to try to identify the effect of the scheduler, and more specifically to the preemptions that a scheduling algorithm brings with itself, on a set of threads running some intensive transactional memory workload. This has been done by running one of the benchmark (sequential access to the elements of a linked list) with different scheduling policies, among the ones available on Linux. Then (in some of the runs), such a policy was altered upon a thread entered a transaction, as well as restored after that was successfully committed. The scheduling policy chosen for the transaction execution was SCHED_FIFO, with priority 50, while the original policy has been varied among SCHED_OTHER, SCHED_RR still with priority 50 or SCHED_RR with priority 40. Note that, in case of SCHED_OTHER original policy, the net effect of changing to SCHED_FIFO while inside a transaction should mimic the disabling of preemption – at least by other SCHED_OTHER thread – and thus potentially bring benefits in terms of determinism and maybe throughput. In order to avoid thread migrations (the effect of which will be investigated in the ongoing work), all the threads of the benchmark were also statically assigned to a CPU.

Figure 17 shows the typical throughput figures (i.e., transactions per second) for the cases of original (non-RT) and real-time enabled (RT) versions of the transactional memory library. When applicable, threads runs at the given scheduling-policy:priority on the left, and transition toward the one on the right of the arrow.

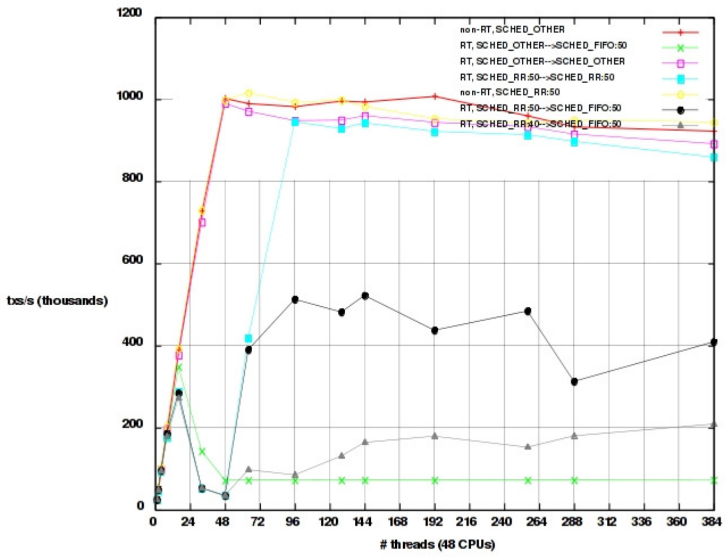


Figure 17: throughput benchmarks for the original (non-RT) and real-time modified (RT) transactional memory library with various scheduling policies.

What the current results mainly show (Figure 17) is that system calls deeply impact on the overall throughput, suggesting that clever methods for making the STM framework communicate with the kernel are worthwhile to investigate. That is the reason why the curves that show better scalability and throughput are the ones where the syscalls are ineffective, i.e., introduce less overhead. On the other hand, as it is possible to see in Figure 18, the disabling of the preemption seems to improve quite a bit the capability of an update transaction to successfully commit, which is something promising from the point of view of real-time and determinism.

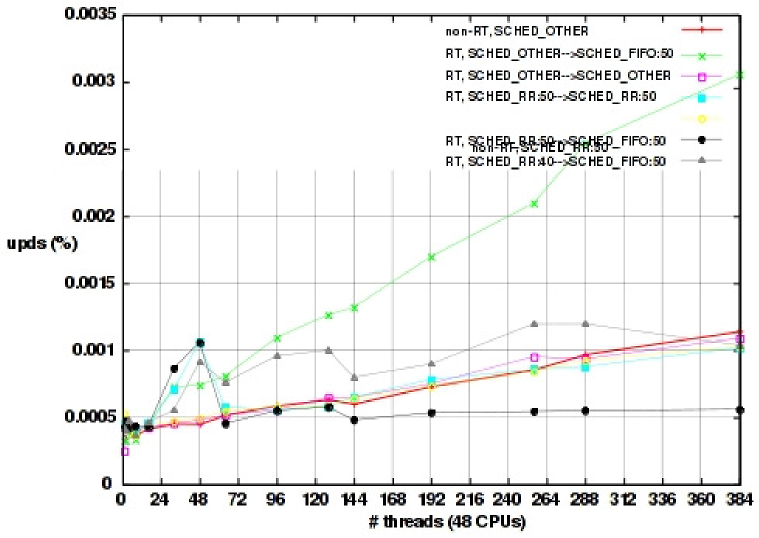


Figure 18: As above, but for % of successful update transactions.

C) Ongoing Work and Proposed Approach

Further investigation on real-time transactional memory will then be carried out during the remainder of the project. The directions toward which this investigation will be look into are mainly two, identifiable as "off-line" and "on-line".

Trying to find out whether it is possible to analyse off-line, from a real-time perspective, a transactional memory based set of threads basically means that if characteristics of the threads, like resource usage patterns, execution times, duration of the critical sections, etc., are given, the actual behaviour could be predicted. In this case, the most challenging aspect definitely is bounding the number of times a thread will need to rollback and restart its transactions because of conflicts, and as a consequence the amount of time a thread spends in these phases, since this directly impact on its execution time. This mostly target the needs of hard real-time systems or, in general, all those systems where confident enough estimations of the parameters are needed and possible. Even in the general case, very pessimistic bounds – e.g., wasted computation time proportional to the number of threads by the number of transactions by the duration of the longest transaction, or similar – could be very simple to derive, but at the same time quite useless. On the other hand, enforcing an ordering in the conflicts resolving logic, and/or a maximum number of times a thread can experience the rollback of one of its transactions would lead to more permissive formulas, and thus to an higher degree of saturation for computational resources in the system, but will also potentially limit the level of concurrent access to transactional data. So the trade-off between these two alternatives needs to be found, and it can be mainly done by means of simulations.

On-line handling of real-time constraints in an STM scenario is specifically meant for contexts where no such knowledge is easily achievable, but still priority (or deadline) ordering among the various activities needs to be preserved as much as possible, since that is what enables at least guaranteed service levels, e.g., some pre-specified amount of CPU execution time (a.k.a., budget) every given time interval (period). In this case, the contention manager of the STM framework plays a crucial role, and two possible strategies for trying to avoid priority inversions can be envisioned, even right now:

1. in case of a conflict, if it is possible to chose which thread to abort, the one with the lower priority (later deadline) should be the one;

2. in case of a conflict, if the thread with the lower priority (later deadline) continues, while the one with the higher priority (earlier deadline) needs to abort the transaction, the priority (deadline) of the former can be boosted to the one of the latter. This is different what has already been proposed in existing STM contention managers (e.g., Eruption [48]), since the priority which is considered is purely the one of the thread running the transaction, not some kind of heuristic estimation of the progress done;

Depending on which (real-time) scheduling algorithm is considered, more specific techniques for limiting the number of preemptions that a thread running inside a transaction – especially if a highly contended one – suffers from needs also to be adopted, e.g., something like the BandWidth Inheritance Protocol [49][50] in case of Resource Reservation. Differently from many of the previous works, the priorities or deadline that will be considered are the one associated to the thread, not some additional information that needs to be added to a transaction. In fact, this is what usually happens in real-time systems, although it is possible to think to the possibility of adding to a transaction extra information for affecting and helping the mechanisms, such as locks have ceilings in the Priority Ceiling Protocol [43]. Moreover, depending on the situation, it is possible to combine 1. and 2. for achieving the best performances in terms of both predictability and throughput.

For all the above to be possible, interactions between the contention-management logic and the underlying OS, to put the priority mangling in place, are mandatory, and thus mechanisms that makes it possible with the lowest possible overhead needs to be investigated or proposed.

3. Speculative Access to Memory

Computing industry relied on increasing the clock frequency and on uni-processor architectural enhancements to improve performance. The situation has changed nowadays, small architectural improvements continue, but power an thermal issues have made the approach of increasing clock frequency non feasible anymore. Semiconductor fabrics continue developing the manufacturing technology and according to the still valid Moore's law, the number of transistor per unit area double every few months. Processors designers use the additional transistor density to place multiple cores in the same die.

This creates a situation in which only multi-threaded applications will take advantage of the new available resources and increase performance. To be able to produce parallel programs is the main concern of the new multicore era.

Many applications exhibit irregular access to memory, a complex control flow and dynamic loop limits. Normal code parallelization and optimization need to take a pessimistic approach to guarantee correctness of execution due to data and control dependencies. Being able to resolve and remove data dependencies in run-time can dramatically improve parallelization.

SAM can potentially widen the scope of single threaded applications that can be parallelized by resolving data dependencies at run-time that otherwise would prevent parallelism from being extracted. SAM is a parallelization technique that is applied to regions of code that might contain a great amount of parallelism but cannot be statically proven to preserve the sequential behaviour under parallel execution. Speculative worker threads execute code out of the sequential order even when these may contain a true data dependencies.

The master thread keeps the correct sequence of state and control flow. While the speculative worker threads, may consume and produce "dirty" values, that is the reason it is necessary to track the inter threads dependencies and memory accesses to revert to a safe point and restart the computation when a dependency violation happens, this is known as a ''flash back recovery''.

To obtain correct execution threads merge their memory changes into the global non-speculative memory space when it can be proven that no dependency violation was generated. The commit is controlled by the non-speculative master thread and that is the reason why they are usually serialized, simulating the behaviour of the sequential program. This can limit the scalability in the number of processors contributing to speed up.

A) APPROACH

In this first approach to develop a framework for accessing speculatively to memory to test its feasibility, it was decided to not used automatic parallelization. SAM was implemented as a library that has to be used by the programmer. That means that the task of identifying the code with high probability of being parallelizable has to be done manually, and it is responsibility of the user of the library to explicitly accept the merges of memory written by the speculatively worker threads. The library will detect the true data dependency violations and assure that the state of all the affected threads is reverted to a previous safe state. The data dependencies are detected within a memory region that can be wider than the used type size, defined by the user. This may cause false positives but at the same time reduce the overhead for each read and/or write to memory. It is the user who has to decide which is the right partitioning size of the memory into regions to obtain a balanced trade between the speculative read/write overhead and the false positives caused.

The library has been written in C as it has direct access to memory addresses, allows to work at address level, and provides good and explicit pointer arithmetic.

TERMS

SAM Context: defines the structure that handles a set of memory regions, a main thread an a set of speculative worker threads.

Memory Region: is the set of continuous memory addresses in which the SAM library is controlling the memory accesses and handle the data dependencies.

Main Thread: is the thread that keeps the correct sequence of state and control flow of the sequential program, it is responsible for the non-speculative memory, to launch the speculative worker threads, and to decide the merging of the memory modified by the worker threads.

Worker Thread: is a thread that accesses to memory speculatively and that is flashed back when a true data dependency violation is detected.

Flash back: is the process that happens when SAM detects the violation of a true data dependency; the worker thread is restored to a previous safe state and none of the memory changes are merged into the "global memory".

SYSTEM DESIGN

This implementation of SAM gives the user a very similar interface to the ''POSIX Threads'' library interface, and it could be said that SAM is comparable to a speculative version of ''POSIX Threads''. All the speculatively processing is realized within a SAM context. A SAM context consists of a main worker thread, a set of memory regions which are declared speculative and a set of speculative worker threads. This allows to create a speculative worker thread that at the same time is the main thread of another SAM context. This implementation was designed to test the concept of speculative access to memory and measure the possible speed ups and find the obstacles and drawbacks of this technology. Those are the reasons the programmability friendliness and ease of use were not taken yet in consideration.

The speculative read and writes are denoted by the user, which greatly reduces the overhead produced by the detection of the violations of true data dependencies, in exchange of an increased programming effort. Each speculative worker thread has its own private memory space for writing into the SAM memory regions. This space is created with the first speculative write. The access to this region (for writing or reading) has to be realized with the use of a special macro *__SAM_ADDR__* that translates the addresses requested from the global shared memory to the private speculative memory space when needed.

When a true data dependency is detected, for example when the main thread writes to a region of memory from which an speculative worker thread has speculatively read, then this worker thread is considered invalid. It's private memory space is cleaned, and its variables and program counter is set up to the initial status of the execution, so that all the processing is recalculated with the new values written by the main thread. This is a critical operation, with a big overhead, but needed to ensure a correct execution of the program. Data dependence violations are detected when they happen which can cut the cost of mis-speculations.

The data dependencies are controlled at memory region level, this means that a RAW dependence (Read after Write) inside the same memory region is always detected as a true dependence even if the read and write were realized to different memory addresses. By using this strategy the overhead of analyzing the memory write and reads is greatly reduce but it has a downside, the creation of many false positives is that many unnecessary flash backs will be generated. The user will have to decide which is the best size of each memory region to obtain the best performance.

All the function calls provided by the SAM library are reentrant and protected against race conditions. Any function can be called from any main or worker thread, and it will not lead to an inconsistent state of the SAM context.

The result of the calculation of an speculative worker thread, and the modifications that it made to its private memory space are retrieved and merged, respectively, when the main thread of the SAM context indicates that there are no going to be more accesses to the memory region and the SAM library detected and solved all data dependencies. The merge of the private memory space into the global share memory space is done sequentially which reduces the scalability in the number of processors contributing to speed up.

CHALLENGES AND OBSTACLES

The main purpose of writing an speculative system is to increase parallelization to obtain a speed up of the execution of the program by keeping the free computing busy doing work that might be needed later. For this, the right equilibrium between the ease of use, overhead by memory access analysis, and cost of a miss prediction (detection of a data dependence violation and its resolution with a flash back) has to be found.

To improve the ease of use, the ideal solution would be to generate an automatic speculative parallelization (done by the compiler and/or the Operating System). This solution has not been considered yet due to the early stage of research of this topic and the high cost of implementing an automatic mechanism. But to keep programmability inside not too difficult limits, and like the current SAM design is relying on the ''POSIX Threads'' interface, the provided interface could be considered a partial ''POSIX Threads'' layer with speculative extensions.

The analysis of access to memory is very expensive from the computing point of view, as for each read or write many assembler instructions have to be executed. To reduce the impact of this phenomena, the program is allowed to access freely to memory, but to make a call to the library to mark a region as read or written. Often, processing is done in arrays of memory, this causes that many times even when a false positive of a data dependency violation is the one forcing the flash back, anyway, the real true positive would have happened a few cycles or iterations later. If this is not the case, to avoid false positive detections (and also avoid useless and costly flash back generation) the registered memory regions in the SAM context should be smaller.

The cost of creating a speculatively private memory space is high (memory has to be reserved, copied, then translate the addresses when accessing to them, and later merging the memory changes into the global shared memory), that is the reason this space is generated on the fist speculatively write. This is a sensible way of proceeding as many speculative worker threads might only read from a certain region of memory and never write to it.

B) Alternative Implementations

HARDWARE:

The SAM architecture relies on keeping a different and separated memory space for each thread. Most of current systems have a single memory space with cache-coherency among the different cores. That is the reason the registered memory regions have to be copied and later merged when a speculative worker thread writes to it. It can clearly be seen that systems with different address spaces would simplify the implementation and decrease the overhead of many of the operations.

KERNEL:

A kernel with speculative memory access control is feasible to be implemented in current systems. The concept of memory region could be mapped to memory page, a memory page becomes a memory region. The advantages of such a system would be that the adaptation of current programs or the implementation of new ones that can use this feature should not require a big programming effort. Also the detection of data dependencies should have a smaller overhead as it is relaying on the CPU hardware MMU system. The disadvantage is that the memory region size would be fixed to the page size, which in same cases will produce an increase on the false positive in the data dependency detection, and with it the number of not necessary flash backs.

C) Future Work

During the implementation and evaluation of the SAM library some new ideas and features that would improve the performance and usability were considered necessary, of which stand out:

- **Thread priority:** A speculative worker thread priority is considered necessary so that *inter-speculative-thread* dependencies can be resolved automatically, and the main worker thread would not have to explicitly and individually merge the memory changes that each worker thread produced in its private memory space.
- **Maximum simultaneous threads:** When more speculative worker threads than resources (cores) are available, then a big performance loss can be caused. This is produced by the fact that the master thread is preempted from the CPU, and this should be avoided for trying to keep maximized the performance of the non speculative part. Trading work done by the main thread for speculative work can only lead to a loss in performance.
- **SAM speculative worker thread "internal scheduler":** The number of threads that are scheduled for running, plus a priority for merging the speculative memory changes could be handled by an in-library software scheduler, that would decide according to the number of resources available, and the priority of each task, which threads should be running at each moment.

D) Early results

This first proof of concept implementation of a Speculative Access to Memory framework helped us to get a precise idea of which are the strong and weak points of this technology. Although by now we have only programmed “synthetic” programs with the aim to test, debug and get the first performance numbers.

Our conclusion is that the overhead caused by the functions of the library create less overhead than expected, and that it is quite easy to obtain some performance improvement. But at the same time the programmer must know well what it is being done because it is very easy to make the system perform worse, for example, with launching more speculative threads that slow down the main thread (that controls the sequential flow).

After writing some examples and tests, so having some experience in using this library, we feel that the use is easy but not trivial and the use is very repetitive but can lead to mistakes if the programmer is not careful. But we consider that, in contrast, the integration inside an automatic parallelization mechanism would be quite straight forward.

Many of these downsides, and design flaws of the current implementation will be solved in the next implementation, currently under design.

V. Code and Resource Migration

The research and development of a component base kernel in WP3 [62] sets the framework for a distributed, service-oriented and lousily coupled operating system. The applications of such framework on execution management is explored in this chapter, in particular component migration, which is applicable to processes and threads.

Migration can be applied to reduce latency by bringing the components closer to where they are being accessed, this would fit within load balancing requirement. It's also applicable has a prevention against faults, there are hardware devices that can detect conditions that will cause a failure in the near future. Of course, migration is not applicable to components that directly manage hardware resources.

Migration has costs in overhead and efficiency, it also causes the temporary unavailability of services, manifested as a latency spike, while the migration is in progress. Migration should, ideally, be a rare event. One way to reduce overhead is to delay the migration of dependent resources with a stub components. Components also have different requirements in terms of data availability. For example, an address-space can be migrated on demand using a core memory management unit capabilities, this is the case for today's operating system memory management strategies.

CONDITIONS

The relationship between components affects the migration process. The most simple case would be migrating a component with no references to other components. In this case, the migration would require only the serialization of the required data. In other case, a component that can have many references to other components can easily make the migration operation very expensive.

Prior to migrating certain resources, that is threads, execution must be suspended prior to the migration operation. Once the migration is complete, execution can is resumed at the new location.

OPTIMIZATION STRATEGIES

The relationship requirements and data dependencies can be exploited to mitigate the overhead of migration.

In the particular case of a process or thread address space, the demand loading enabled by the core paging unit can be exploited for this purpose. Further, profiling the access behaviour can enable optimal migration in the background. The approach presented here requires coordination with a memory manager, thus it is custom solution.

In the case of a whole process migration, a segmented approach can be exploited based on the existing data dependencies. With this strategy, migration can be done in individual stages to minimize the time the process execution is suspended.

To mitigate the overhead of migration in the general cases, we propose a cost analysis using priority classification. Thus dependencies are migrated on priority basis. Referenced component instances can be thread local, process local or global. At least the last case is mapped with stubs. But for efficiency, thread local component instances should be migrated. However, the number of thread local objects can be highly dynamic. Thus dependencies are migrated on a priority basis until a maximum cost is achieved, for the remaining instances they will be mapped with stubs and latter migrated lazily in the background.

VI. References

[1] *Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. “*Stretching transactional memory.” In PLDI ’09.

[2] *Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun.* "STAMP: Stanford Transactional Applications for Multi-Processing." In IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization, 2008.

[3] *Rachid Guerraoui, Michal Kapalka, and Jan Vitek.* "STMBench7: A benchmark for software transactional memory." In EuroSys ’07.

[4] *Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. “Software Transactional Memory: Why Is It Only a Research Toy?”.* In ACM Queue, September 2008.

[5] *Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui.* “Why STM can be more than a Research Toy.” In CACM, April, 2011.

[6] *Transactional Memory Specification Drafting Group.* “Draft Specification of Transactional Language Constructs for C++.” Version 1.0, August 2009.

[7] *Maurice Herlihy and Eric Koskinen.* "Transactional boosting: a methodology for highly-concurrent transactional objects." In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, 2008.

[8] *G. Bollella, J. Gosling.* “The real-time specification for java.” Computer 33 (2000) 47–54.

[9] *A. Burns, A. Wellings.* “Concurrent and Real-Time Programming in Ada 2005.” Cambridge University Press, 2007.

[10] IEEE. “Information Technology -Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions.” 2004.

[11] *J. W. S. Liu, K.-J. Lin, R. Bettati, D. Hull, A. Yu.* “Foundations of Dependable Computing.” volume 284 of The International Series in Engineering and Computer Science, Springer US, pp. 157–182.

[12] *D. Hull, W. chun Feng, J. W. Liu.* “Operating system support for imprecise computation, in: In Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities.” pp. 96–99.

[13] *A. Quagli, D. Fontanelli, L. Greco, L. Palopoli, A. Bicchi.* “Designing real-time embedded controllers using the anytime computing paradigm.” pp. 1 –8.

[14] *T. Cucinotta, D. Faggioli, A. Evangelista.* "Exception-based Management of Timing Constraints Violations for Soft Real-Time Applications." In proceedings of the 5th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Dublin (Ireland), July 2009.

[15] *Tommaso Cucinotta, Dario Faggioli.* “An Exception Based Approach to Timing Constraints Violations in Real-Time and Multimedia Applications.” In proceedings of the 5th International IEEE Symposium on Industrial Embedded Systems (SIES) 2010, Trento (Italy), July 2010.

[16] More information available at: www.mplayerhq.hu

[17] More information available at: http://www.bigbuckbunny.org/

[18] More information available at: http://weather.ou.edu/~apw/projects/stress/

[19] IEEE Standard for Information Technology. “Portable Operating System Interface (POSIX).” http://www.opengroup.org/onlinepubs/009695399.

[20] OCERA European Project (Open Components for Embedded Real-time Applications), http://www.ocera.org.

[21] *Dario Faggioli, Marko Bertogna, Fabio Checconi.* “Sporadic Server Revisited.” Proceedings of 25th ACM Symposium On Applied Computing, Sierre, Switzerland, March 2010.

[22] *Mark Stanovich , Theodore P. Baker , An-I Wang , Michael Gonzalez Harbour.* “Defects of the POSIX Sporadic Server and How to Correct Them.” Technical Report, October 2009.

[23] *Dario Faggioli, Antonio Mancina, Fabio Checconi, Giuseppe Lipari.* “Design and Implementation of a POSIX compliant Sporadic Server for the Linux Kernel.” Proceedings of the 10th Real-Time Linux Workshop, pp. 65 – 80, Colotlán, Jalisco, Mexico, November 2008.

[24] *R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa.* "Resource kernels: A resource-centric approach to real-time and multimedia systems." In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, 1998.

[25] *Luigi Palopoli, Tommaso Cucinotta, Luca Marzario, Giuseppe Lipari.* "AQuoSA - Adaptive Quality of Service Architecture." Software: Practice and Experience, April 2008, doi 10.1002/spe.883

[26] http://www.ocera.org/download/documents/documentation/wp1.html OCERA Project Deliverable D1.1 - RTOS

[27] *S. Oikawa, R. Rajkumar.* "Portable RK: a portable resource kernel for guaranteed and enforced timing behavior." Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium, Vancouver, BC, 1999.

[28] *Tommaso Cucinotta.* "Access Control for Adaptive Reservations on Multi-user Systems,"14th IEEE Real-Time and Embedded Technology and Applications Symposium, St. Louis, MO, United States, April 2008.

[29] *Jochen Setz.* "Inter-Process Communication in RTAI and RTLinux." Saarland University.

[30] *Dario Faggioli.* "POSIX SCHED_SPORADIC implementation for tasks and groups." available on-line at: http://lwn.net/Articles/293547/.

[31] Website of the European Project No. IST-2001-35102 OCERA – Open Components for Embedded Real-time Applications, available on-line at: http://www.ocera.org.

[32] Website of the European Project No. FP6/2005/IST/5-034026 FRESCOR – Framework for Real-time Embedded Systems based on ContRacts, available on-line at: http://www.frescor.org.

[33] James Litchfield, The Foundations of Solaris Realtime, May 2007, Available on-line at: http://blogs.sun.com/thejel/entry/the_foundations_of_solaris_realtime

[34] IEEE. Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions, 2004

[35] *Fabio Checconi, Tommaso Cucinotta, Dario Faggioli, Giuseppe Lipari.* "Hierarchical Multiprocessor CPU Reservations for the Linux Kernel," in Proceedings of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009), Dublin, Ireland, June 2009.

[36] *Tommaso Cucinotta, Gaetano Anastasi, Fabio Checconi, Dario Faggioli, Kleopatra Kostanteli, Antonio Cuevas, Dominik Lamp, Söeren Berger, Manuel Stein, Thomas Voith, Lars Fuerst, Darren Golbourn, Malcolm Muggeridge.* "IRMOS Deliverable D6.4.2 – WP 6 Execution Environment - Final Version of Realtime Architecture of Execution Environment." January 2010.

[37] *Tommaso Cucinotta and Fabio Checconi.* "The IRMOS Real-Time Scheduler." August 2010. Available on-line at: http://lwn.net/Articles/398470/.

[38] *Paul Menage, Paul Jackson, Christoph Lameter.* "CGROUPS." Available on-line at: http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[39] *Tommaso Cucinotta, Giuseppe Lipari, Dario Faggioli, Fabio Checconi and Sunil Kumar, Rui Aguiar, João Paulo Barraca, Bruno Santos and Javad Zarrin, Jan Kuper and Christiaan Baaij, Lutz Schubert and Hans-Martin Kreuz, Vincent Gramoli.* "State of the Art – S(o)OS Project Deliverable D5.1." July 2010.

[40] *Nicola Serreli, Giuseppe Lipari, Enrico Bini.* "The Distributed Deadline Synchronization Protocol for Real-Time Systems Scheduled by EDF." In Proceedings of RTSS 2010.

[41] *Giuseppe Lipari, Enrico Bini.* "A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation." In Proceedings of ETFA 2010.

[42] *Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bh, Emmett Witchel.* "TxLinux: Using and Managing Hardware Transactional Memory in an Operating System." In Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles.

[43] *L. Sha, R. Rajkumar, J. P. Lehoczky.* "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." IEEE Transactions on Computers, Vol. 39, No. 9. (1990), pp. 1175-1185.

[44] *Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, Gilles Muller.* "Scheduling support for transactional memory contention management." In proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming.

[45] *Walther Maldonado, Patrick Marlier, Pascal Felber, Julia Lawall, Giller Muller and Etienne Rivière.* "Deadline-Aware Scheduling for Software Transactional Memory." To appear in the Proceedings of

the 41th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'11), June 2011.

[46] *Martin Schoeberl, Bent Thomsen, Lone Leth Tomsen.* "Towards Transactional Memory for Real-Time Systems." TU Wien Technical Report 19/2009.

[47] *T. Sarni, A. Queudet, P. Valduriez.* "Real-Time Support for Software Transactional Memory." In proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09.

[48] *William N. Scherer III , Michael L. Scott.* "Advanced Contention Management for Dynamic Software Transactional Memory." In proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC 2005), Las Vegas, NV, July 2005.

[49] *G. Lamastra, G. Lipari, L. Abeni.* "A Bandwidth Inheritance Algorithm for Real-Time Task Synchronization in Open Systems." In Proceedings of the IEEE Real-Time Systems Symposium, London (UK) Dec. 2001.

[50] *Dario Faggioli, Giuseppe Lipari, Tommaso Cucinotta.* "The Multiprocessor BandWidth Inheritance Protocol." In proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS), Brussels (Belgium), July 2010.

[51] *Pascal Felber, Vincent Gramoli, Rachid Guerraoui.* "Elastic Transactions." In Proceedings of the 23rd International Symposum on Distributed Computing (DISC'09).

[52] *Christos H. Papadimitriou, Kenneth Steiglitz.* "6.1 The Max-Flow, Min-Cut Theorem". In Combinatorial Optimization: Algorithms and Complexity, Dover, 1998, pp. 120–128

[53] *Mechthild Stoer, Frank Wagner* “A Simple Min-Cut Algorithm”, Journal of the ACM, vol. 44, No.4, July 1997, pp.585-591

[54] Asanovic, K. R. (2006). *The Landscape of Parallel Computing*. Berkeley: University of California.

[55] Boeijink, A. P. (2010). I*ntroducing the PilGRIM: a Pipelined Processor for Executing Lazy Functional Languages*. 22nd Symposium on Implementation and Application of Functional Languages (IFL) (to appear).

[56] Candaele, B., & Vanmeerbeeck, G. (2011). *MPSoC Hardware/Software Architectural and Design Challenges/Solutions*. Grenoble, France: Design, Automation and Test Europe (DATE) Tutorial Notes.

[57] Lee, E. A. *The Problem with Threads*. Berkeley: University of California, 2006.

[58] Naylor, M. C. *The Reduceron Reconfigured*. International Conference on Functional Programming, 2010

[59] Naylor, M. C.. *The Reduceron: Widening the Von Neumann Bottleneck for Graph Reduction Using an FPGA*. In 19th Symposium on Implementation and Application of Functional Languages, 2007, pp. 129-146.

[60] Peyton Jones, S. *Beautiful Concurrency. Cambridge: Microsoft Research*. 2006

[61] Sutter, H. *The Free Lunch is Over: a Fundamental Turn Toward Concurrency in Software*. In Dr. Dobb's Journal, 2005

[62] *Bruno Santos, Javad Zarrin, Christiaan Baaij, Vicent Gramoli, Aleksander Dragojevic, Tomasso Cucinotta.* "First Implementation Set: Protocols - Project Deliverable D3.2."