Parallelization of the AliRoot event-reconstruction by using a source-to-source transformation

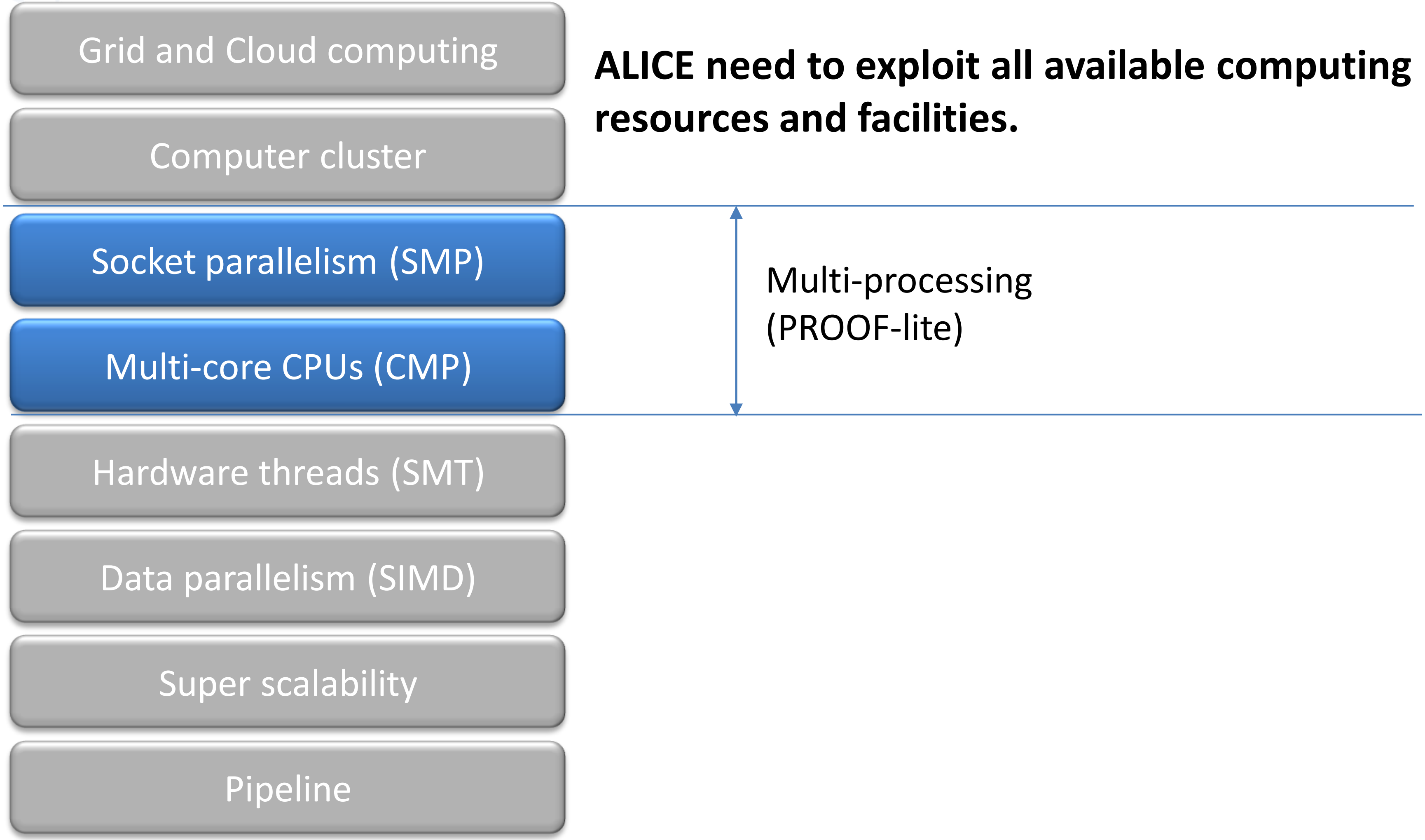
Stefan B. Lohn, CERN

May 24th 2012

CHEP 2012

Introduction

ALICE need to exploit all available computing resources and facilities.



Grid and Cloud computing

Computer cluster

Socket parallelism (SMP)

Multi-core CPUs (CMP)

Hardware threads (SMT)

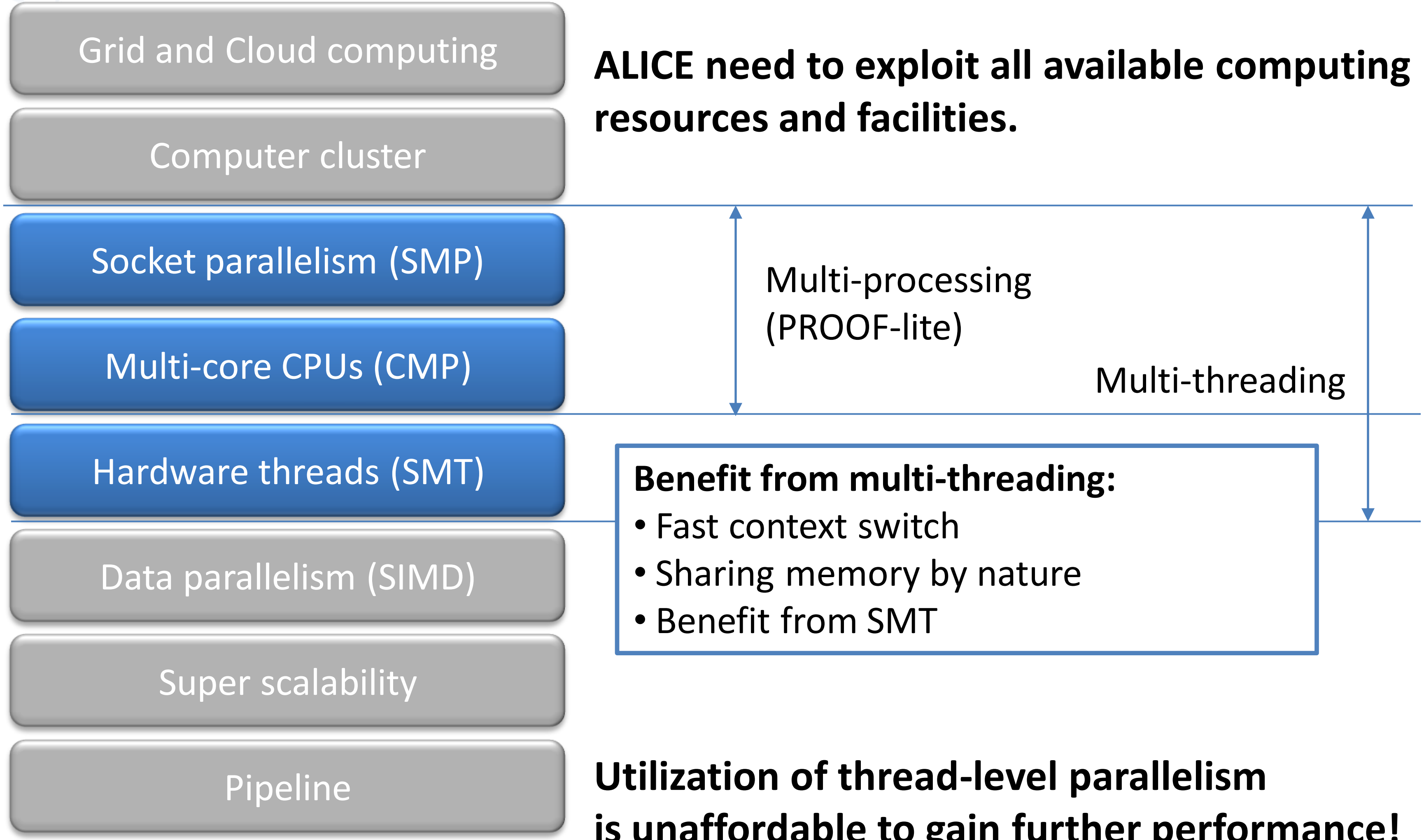
Data parallelism (SIMD)

Super scalability

Pipeline

Multi-processing
(PROOF-lite)

Introduction



- Grid and Cloud computing
- Computer cluster
- Socket parallelism (SMP)
- Multi-core CPUs (CMP)
- Hardware threads (SMT)
- Data parallelism (SIMD)
- Super scalability
- Pipeline

ALICE need to exploit all available computing resources and facilities.

Multi-processing
(PROOF-lite)

Multi-threading

Benefit from multi-threading:
- Fast context switch
- Sharing memory by nature
- Benefit from SMT

Utilization of thread-level parallelism is unaffordable to gain further performance!

Objectives

Remaining issues:

- Thread-safety is difficult to introduce
- Parallel programming is labor intensive

A semi-automatic source-to-source transformation.

(as investigated from the Geant4-MT Project)

Objectives

Remaining issues:

- Thread-safety is difficult to introduce
- Parallel programming is labor intensive

A semi-automatic source-to-source transformation. ?

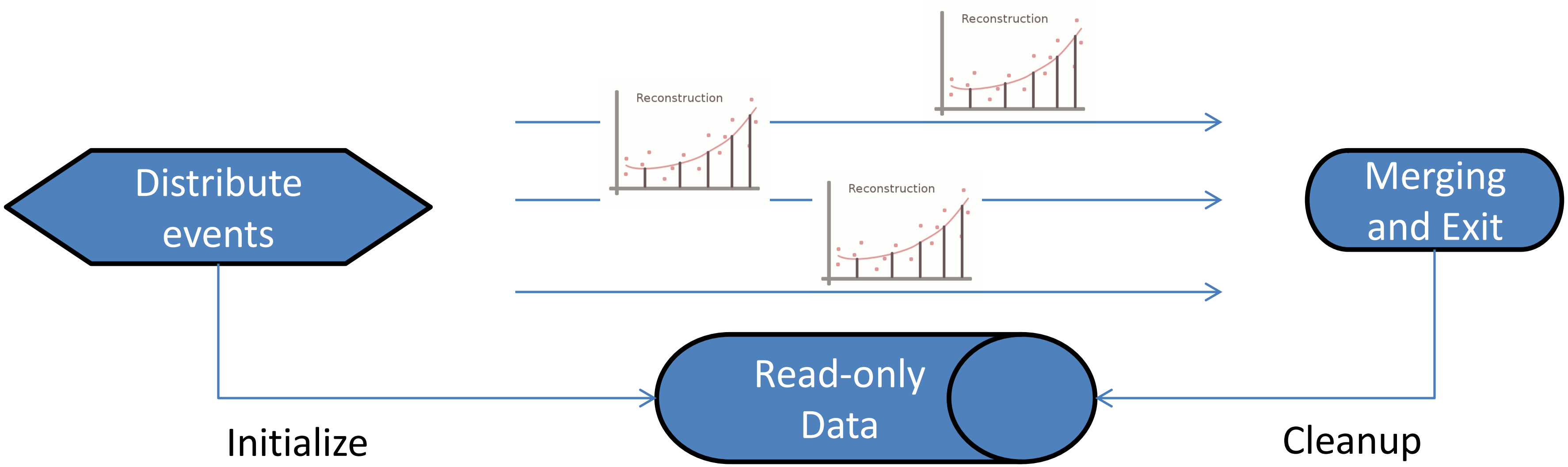
(as investigated from the Geant4-MT Project)

Benefits:

- Faster development process
- No additional development branch
- Without expert knowledge
- User maintain only their sequential code

Objectives

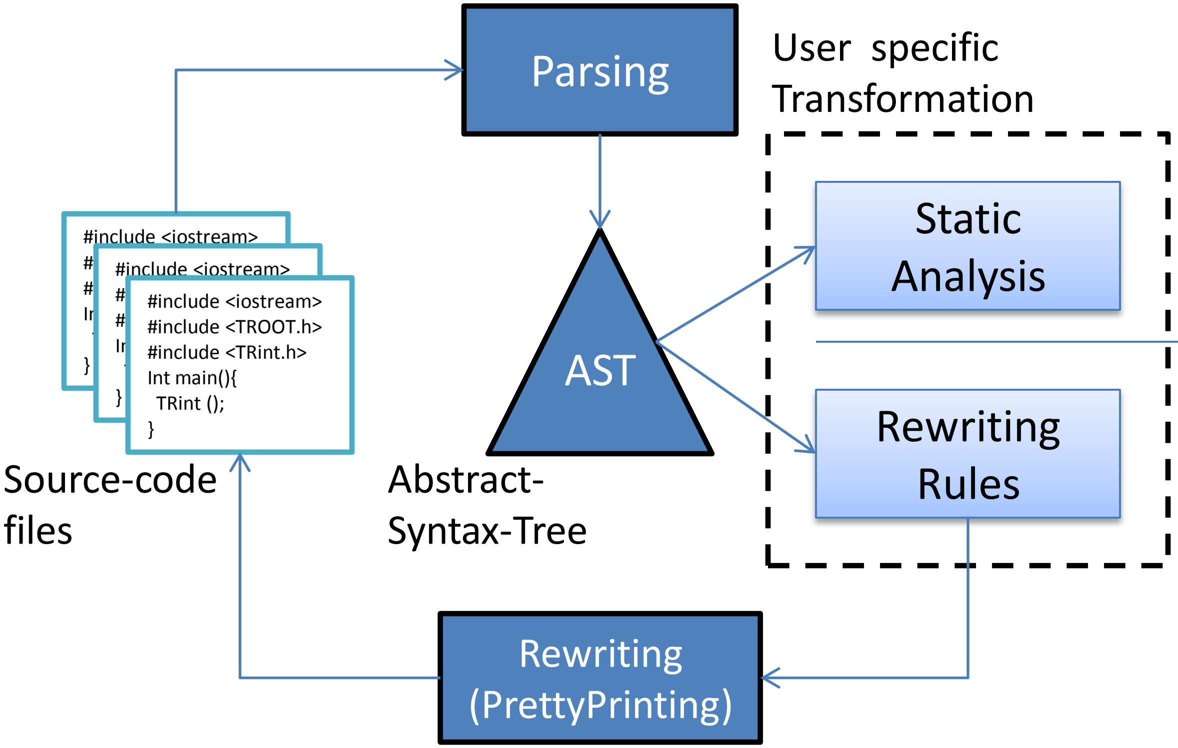
Parallel design:



- Full reconstruction per thread
- No interference amongst threads
- Event-level parallelism

Source-to-Source Transformation

Scheme of the source-to-source transformation:



Parsing

User specific Transformation

```
#include <iostream>
#include <TROOT.h>
#include <TRint.h>
Int main(){
  TRint ();
}
```

Source-code files

AST

Abstract-Syntax-Tree

Static Analysis

Rewriting Rules

Rewriting (PrettyPrinting)

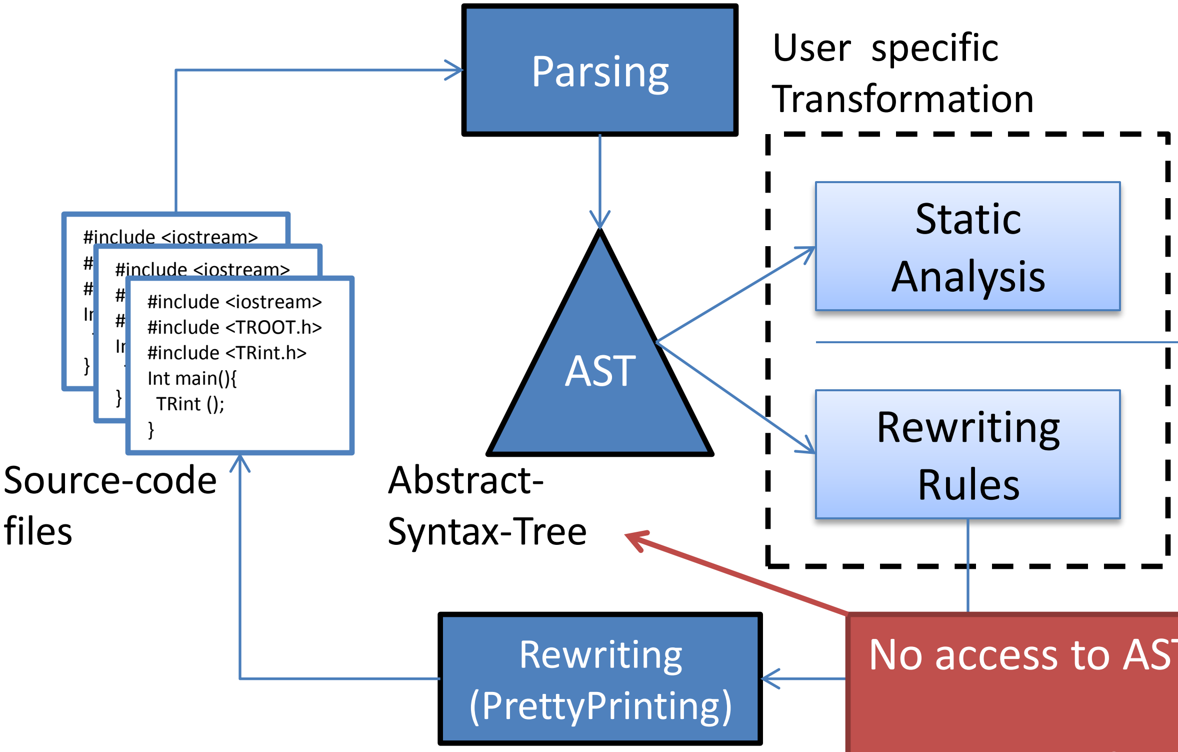
Original Tool:

Augmented GCC parser to return positioning information.

Case-based fact extraction from a token-stream generated by ELSAs parser.

Source-to-Source Transformation

Scheme of the source-to-source transformation:



```
#include <iostream>
#include <TROOT.h>
#include <TRint.h>
Int main(){
  TRint ();
}
```

Original Tool:

Augmented GCC parser to return positioning information.

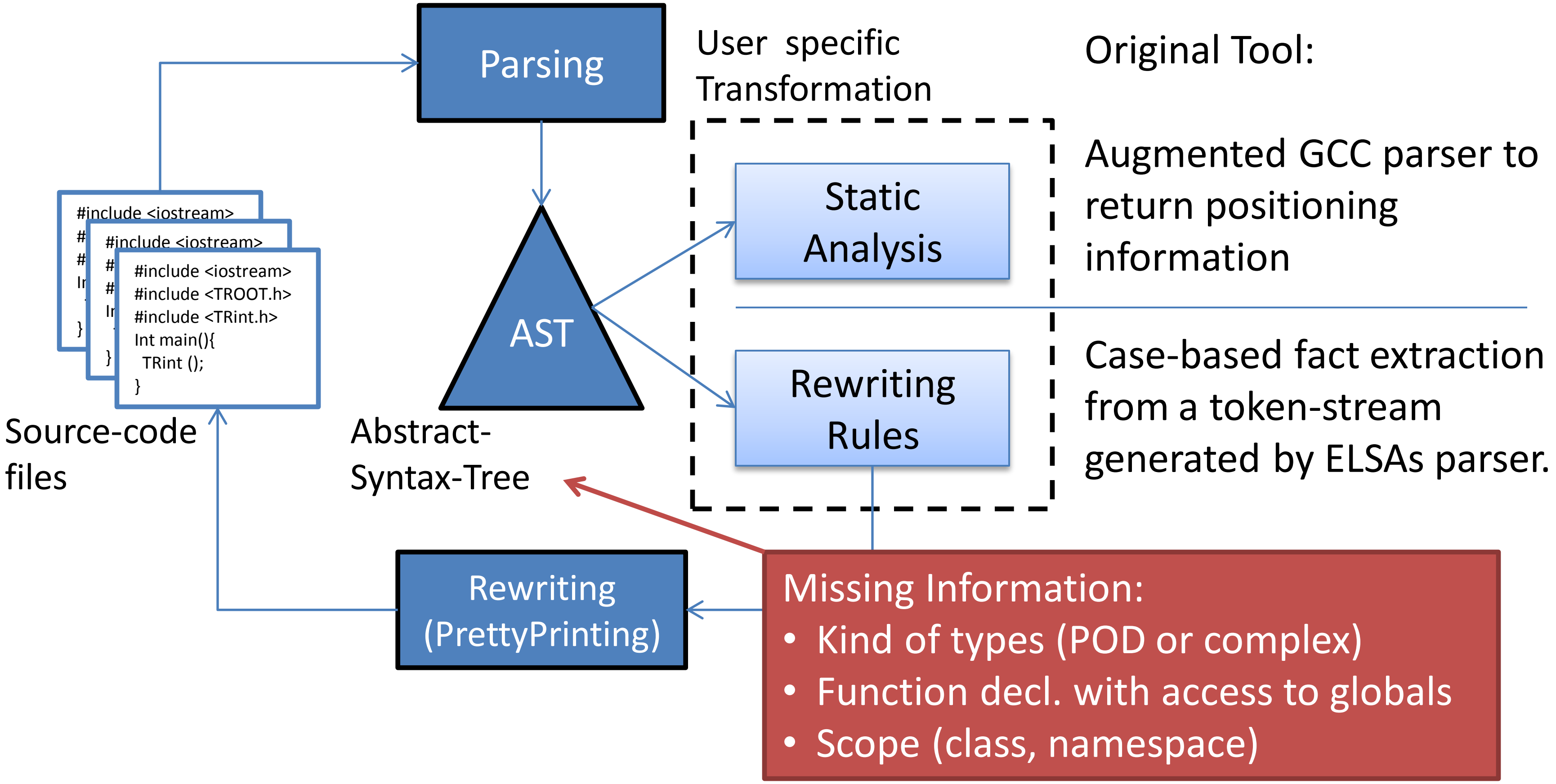
Case-based fact extraction from a token-stream generated by ELSAs parser.

No access to AST during rewriting!

Error prone for heterogeneous source-code like in ROOT and AliRoot
and hard to fix.

Source-to-Source Transformation

Scheme of the source-to-source transformation:



Parsing

User specific Transformation

Original Tool:

Static Analysis

Augmented GCC parser to return positioning information

AST

```
#include <iostream>
#include <TROOT.h>
#include <TRint.h>
Int main(){
  TRint ();
}
```

Source-code files

Abstract-Syntax-Tree

Rewriting Rules

Case-based fact extraction from a token-stream generated by ELSAs parser.

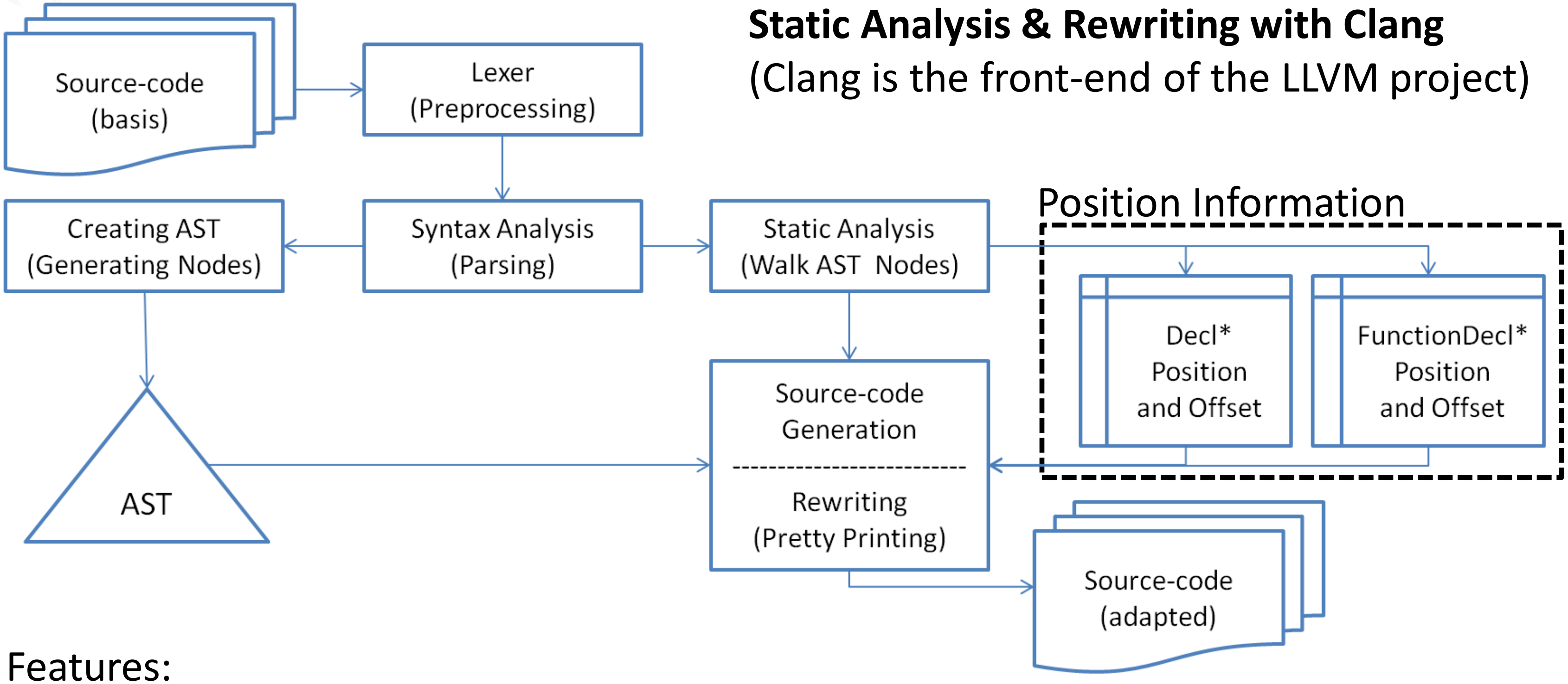
Rewriting (PrettyPrinting)

Missing Information:

- Kind of types (POD or complex)
- Function decl. with access to globals
- Scope (class, namespace)

Source-to-Source Transformation

Static Analysis & Rewriting with Clang
(Clang is the front-end of the LLVM project)



Features:

-> Analysis and rewriting by the same software
-> Access to AST during rewriting
-> Fixes all bugs mentioned and further small problems

1) TTS

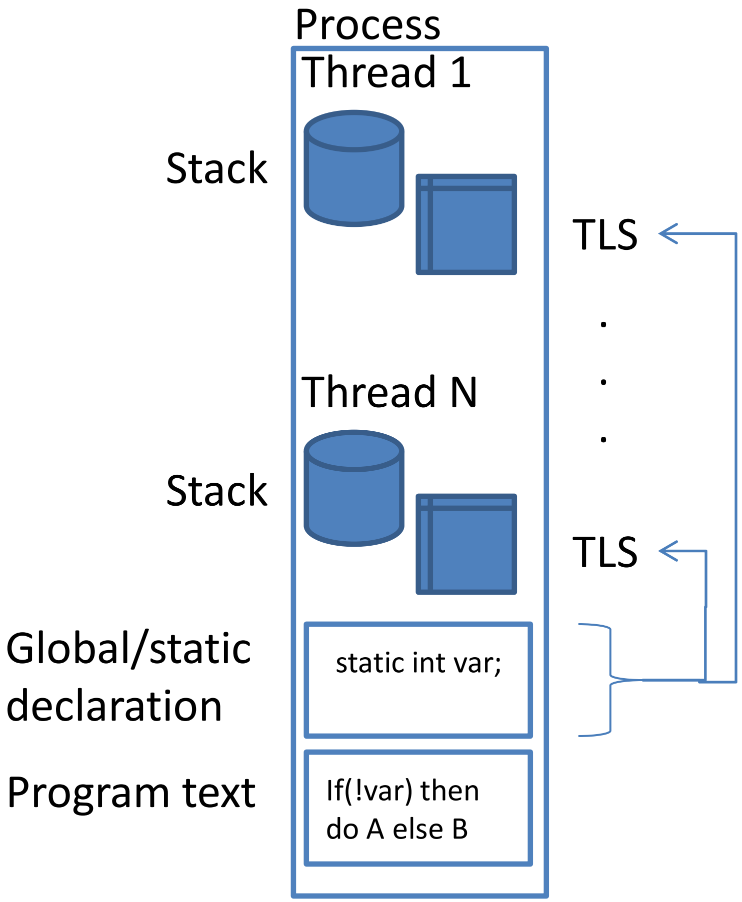
The source-code adaption follows 4 important steps:

	Manual Intervention		Automated
1)		TTS	Privatization of global and static declaration
2)	Selection of sharable classes	TMR	Source-code adaption
3)	Data-race detection (Debugging)	Verification	Protection for unintended write access to read-only data
4)	Customized code optimization	Optimization	

Manual intervention is following a specific procedure to adapt source-code. If it is conducted once, it can be automated for further transformations.

1) TTS

Thread-safety: Access to resources do not lead to unintended interference amongst threads. The strongest kind is *unconditional thread-safety*, where there is no interference at all.



Process

Thread 1

Stack

TLS

.
.
.

Thread N

Stack

TLS

Global/static declaration

static int var;

Program text

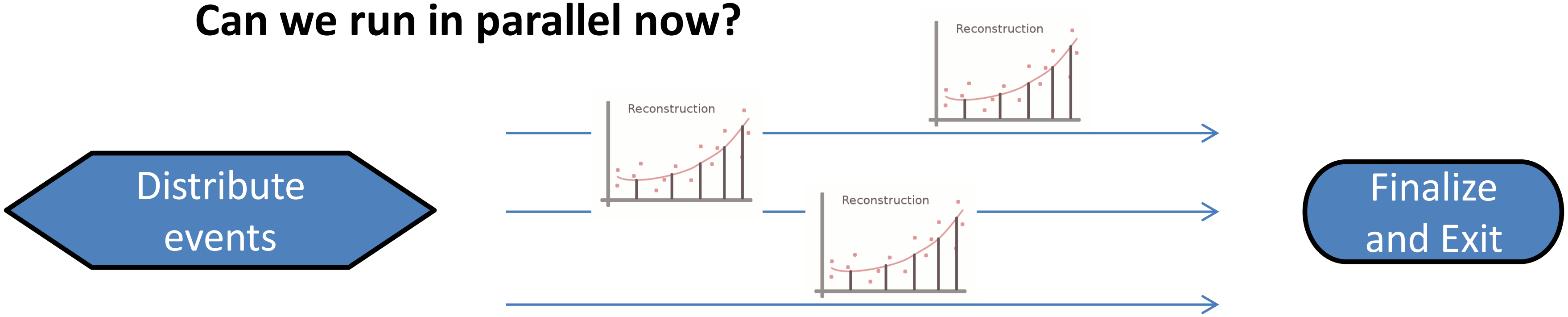
If(!var) then
do A else B

1. Recognize concerned declarations:
   The *recognition* phase is part of the static analysis. The aim is to detect global and static declarations and to collect position information.

2. Privatize recognized declarations:
   For *privatization* the thread-specifier is used to allocate the recognized declarations into thread-local storage (TLS) and to make them private to threads.

1) TTS for ROOT & AliRoot

Can we run in parallel now?



Distribute events

Finalize and Exit

Found (Changed)	Statics	Globals	Extern
AliRoot	2023 (417)	138 (11)	159 (31)
ROOT	3652 (620)	191 (150)	125 (46)
CINT	228 (0)	69 (0)	308 (0)

Only involved soure-code has been addressed.

In practice: **Violations** by

1. source-code generation (global decl. in CInt)
2. external libraries and non re-entrant functions.
3. common accessible resources.

2) TMR

The source-code adaption follows 4 important steps:

	Manual Intervention		**Automated**
1)		TTS	Privatization of global and static declaration
2)	Selection of sharable classes	TMR	Source-code adaption
3)	Data-race detection (Debugging)	Verification	Protection for unintended write access to read-only data
4)	Customized code optimization	Optimization	

Manual intervention is following a specific procedure to adapt source-code. If it is conducted once, it can be automated for further transformations.

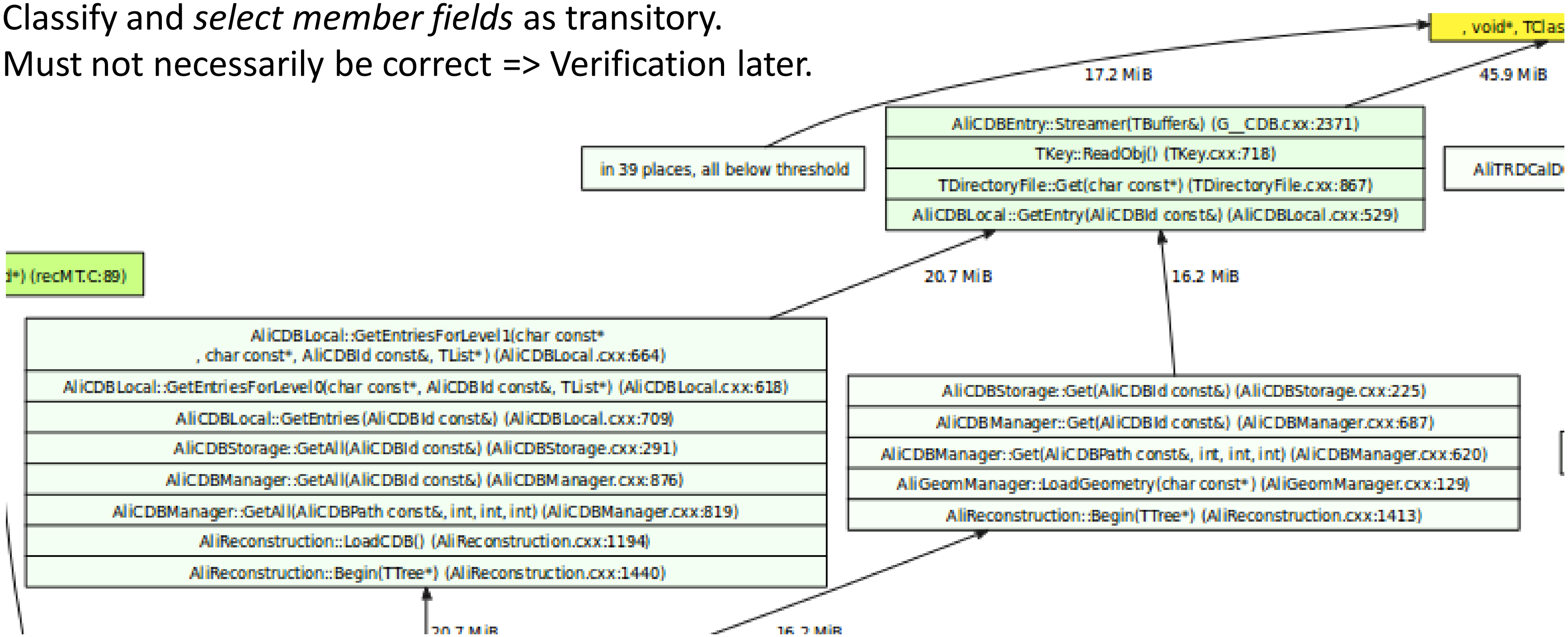
2) TMR

Scheme of the **transformation for memory footprint reduction (TMR)**:

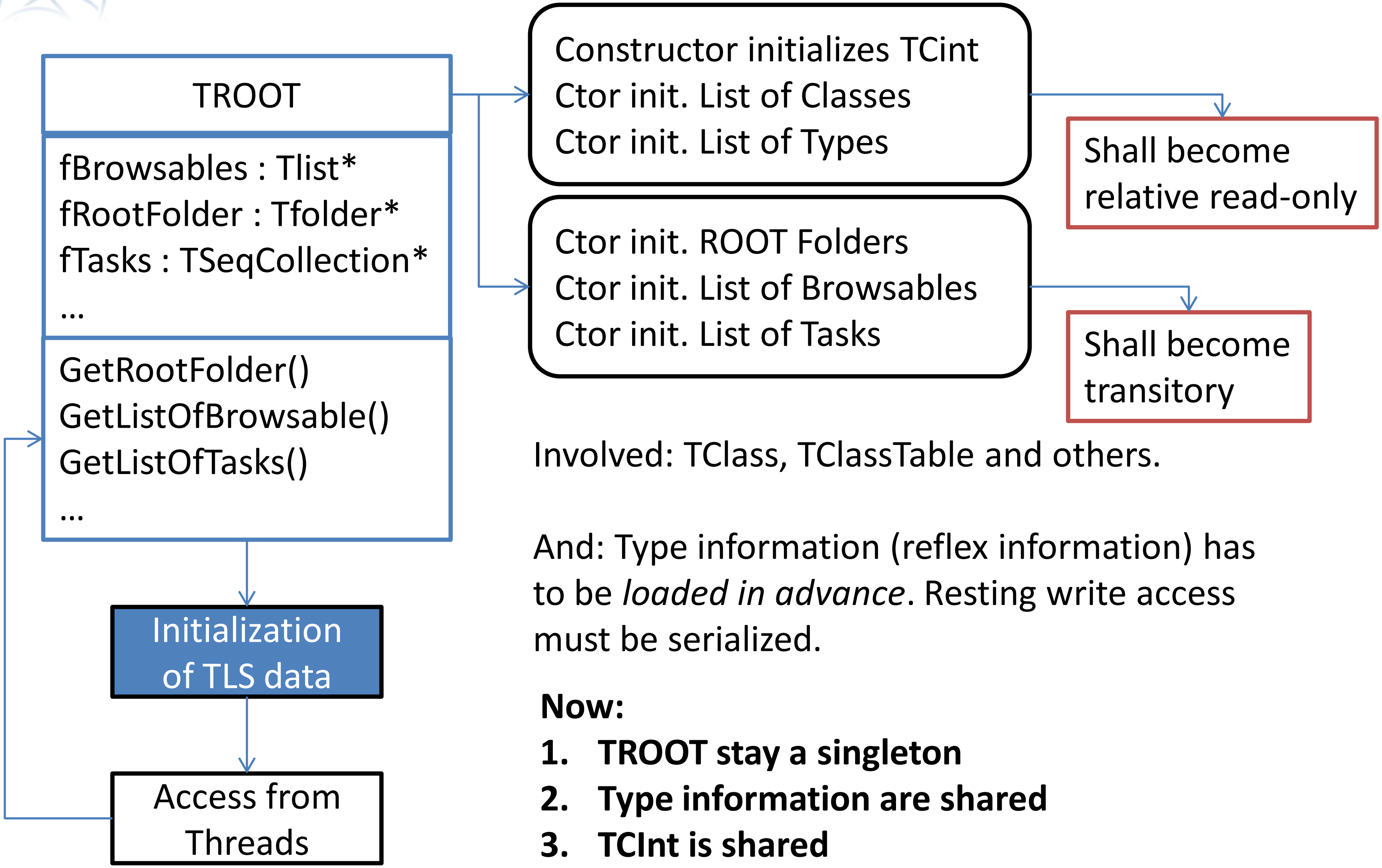
Sharing Classes: Since it is uncertain to find whole classes with static content that can be shared, single member fields can be classified as ***transitory*** and will become thread-local in the transformed source-code. Resting members are assumed to be **relative *read-only***.

Manual

1. Run *memory profiling* to find interesting objects. (e.g. Memory graph of massif)
2. Classify and *select member fields* as transitory.
3. Must not necessarily be correct => Verification later.



2) TMR to share CInt



TROOT

fBrowsables : Tlist*
fRootFolder : Tfolder*
fTasks : TSeqCollection*
…

GetRootFolder()
GetListOfBrowsable()
GetListOfTasks()
…

Initialization of TLS data

Access from Threads

Constructor initializes TCint
Ctor init. List of Classes
Ctor init. List of Types

Shall become relative read-only

Ctor init. ROOT Folders
Ctor init. List of Browsables
Ctor init. List of Tasks

Shall become transitory

Involved: TClass, TClassTable and others.

And: Type information (reflex information) has to be *loaded in advance*. Resting write access must be serialized.

Now:

1. **TROOT stay a singleton**
2. **Type information are shared**
3. **TCInt is shared**

3) Verification

The source-code adaption follows 4 important steps:

	Manual Intervention		**Automated**
1)		TTS	Privatization of global and static declaration
2)	Selection of sharable classes	TMR	Source-code adaption
3)	Data-race detection (Debugging)	Verification	Protection for unintended write access to read-only data
4)	Customized code optimization	Optimization	

Manual intervention is following a specific procedure
to adapt source-code. If it is conducted once, it can be automated
for further transformations.

3) Verification

Which violation of correctness can occur?

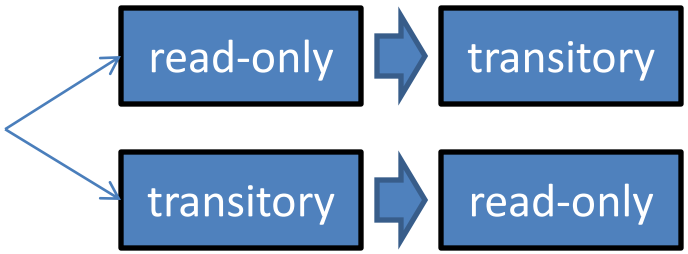
1) Race condition, because of resting global state
2) Race condition, if read-only data are written
3) Access to common resources (process resources, e.g. files, sockets, locks)
4) Semantic relations between resources and global decl.

Solutions:

- 1) & 2) can be detected using dynamic data-race detection tools like **helgrind**.
- 2) can easier be detected with the **MWP Tool**
- 3) can whether be detected by race detection or by **static analysis**
- Semantic relations can be detected by **Bisimulation**

3) Verification

Verification of TMR:



Actual

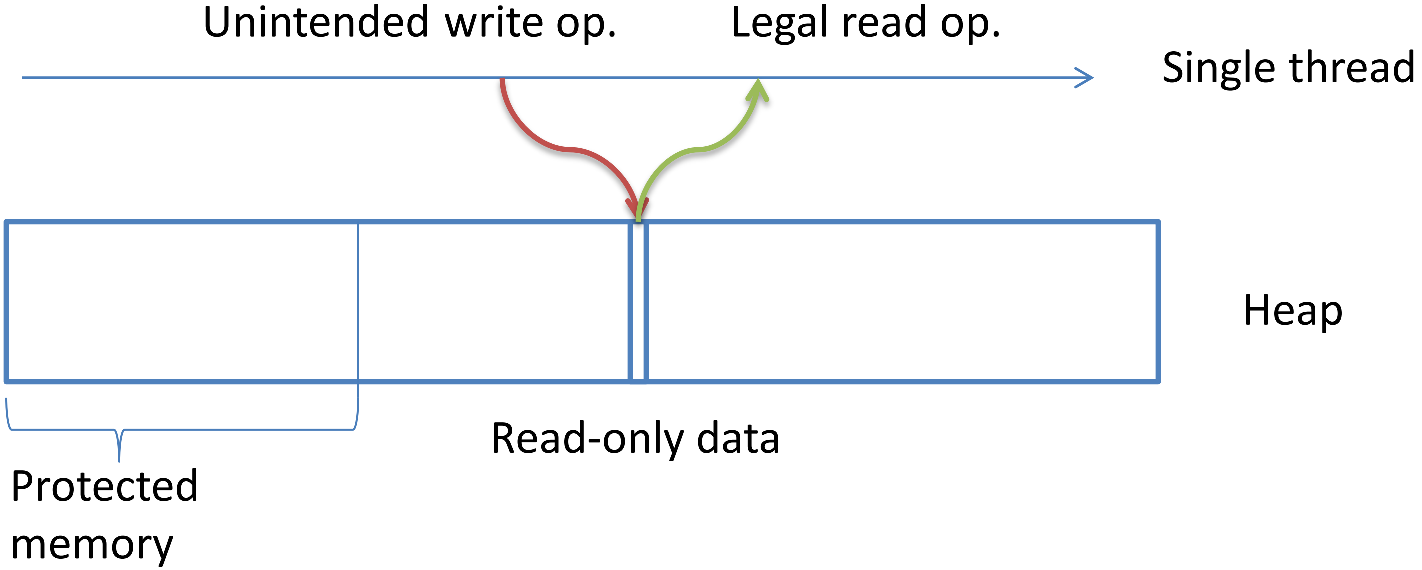
False classification

Class member

read-only → transitory No violation

transitory → read-only **Violation!**

Memory Write Protection Tool (MWPT):



Unintended write op.

Legal read op.

Single thread

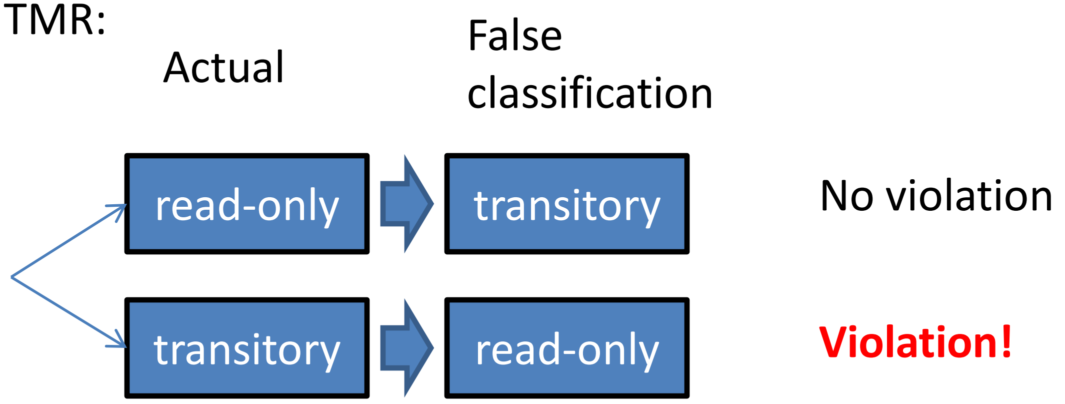
Heap

Read-only data

Protected memory

3) Verification

Verification of TMR:



Actual

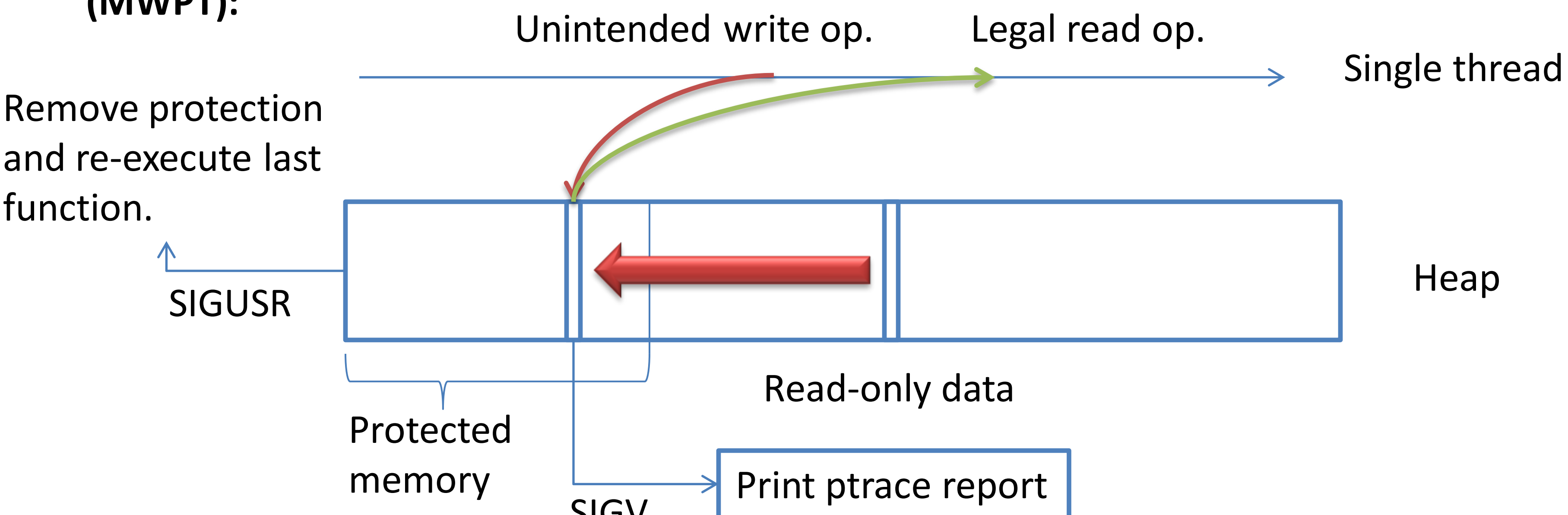
False classification

Class member

read-only → transitory No violation

transitory → read-only **Violation!**

Memory Write Protection Tool (MWPT):



Unintended write op.

Legal read op.

Single thread

Remove protection and re-execute last function.

SIGUSR

Heap

Read-only data

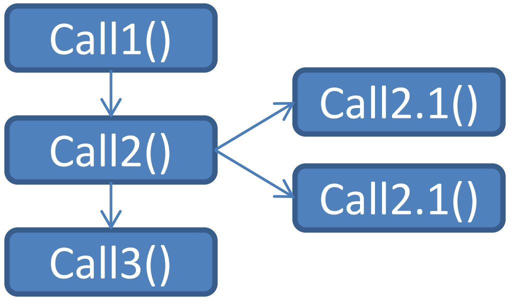
Protected memory

SIGV

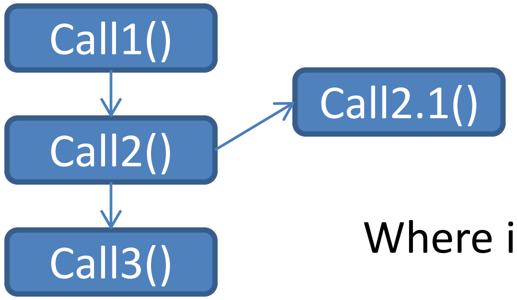
Print ptrace report

3) Verification

Misbehaviour:



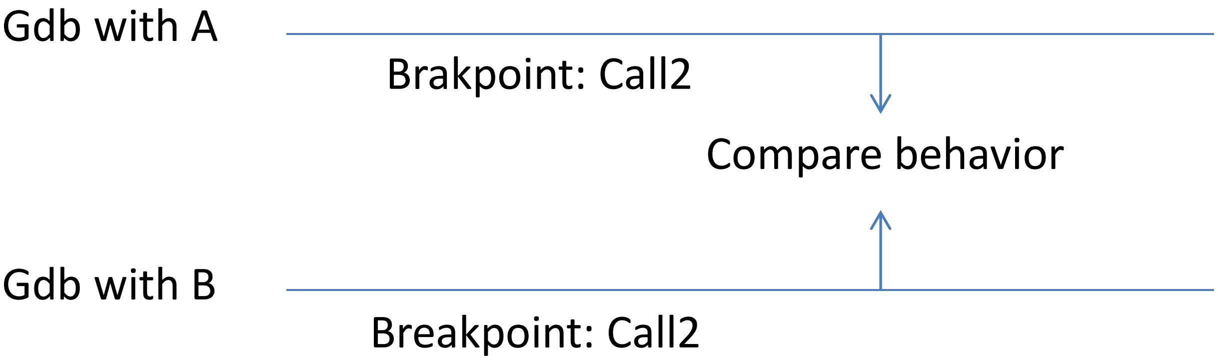
A: Sequential (original) code



Where is 2nd (2.2) call?

B: Multi-threaded (adapted) code

Bisimulation:



Gdb with A

Brakpoint: Call2

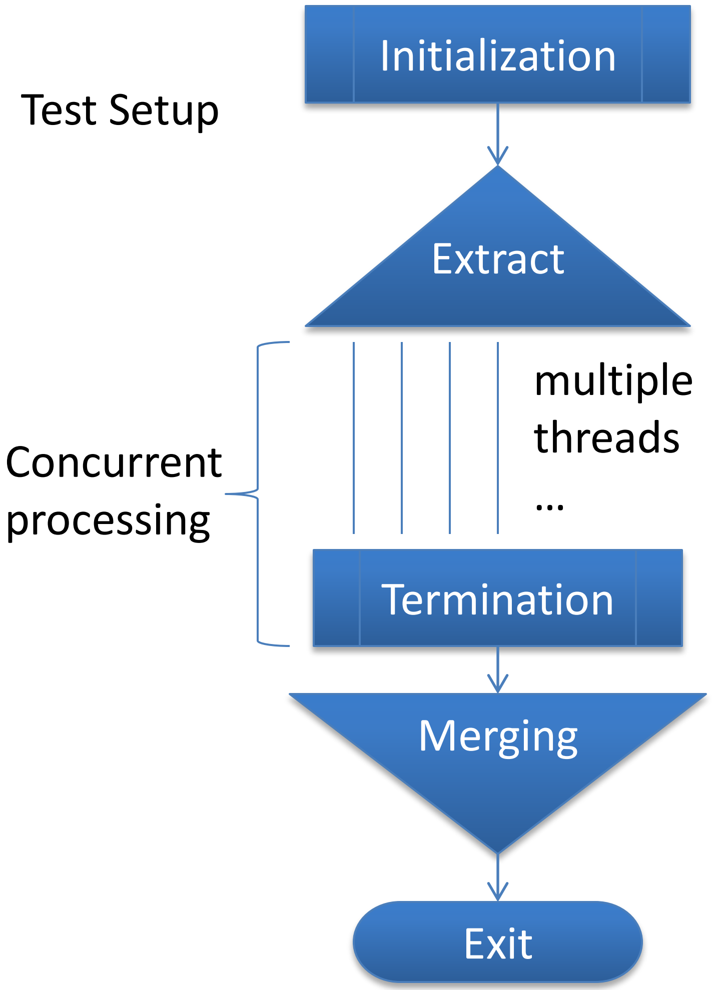
Compare behavior

Gdb with B

Breakpoint: Call2

What about performance ?

Testing Event Reconstruction



Initialization: Divided into 4 steps

1. Creation of reflex data in advance.
2. Preparation of data for shared classes.
3. Per thread: Instantiation of global declaration.
4. Per thread: Initialization of Reconstruction.

Extraction & Merging: Prepare data by loading, extracting and distributing them to the reconstruction threads. Merge the resulting files per threads in the end.

Termination & Exit: During exit objects are going to be deleted, object in container must be cleaned, what is now executed during termination for thread dedicated resources.

Concurrent Processing: Parallel event reconstruction. Currently no master-worker paradigm used.

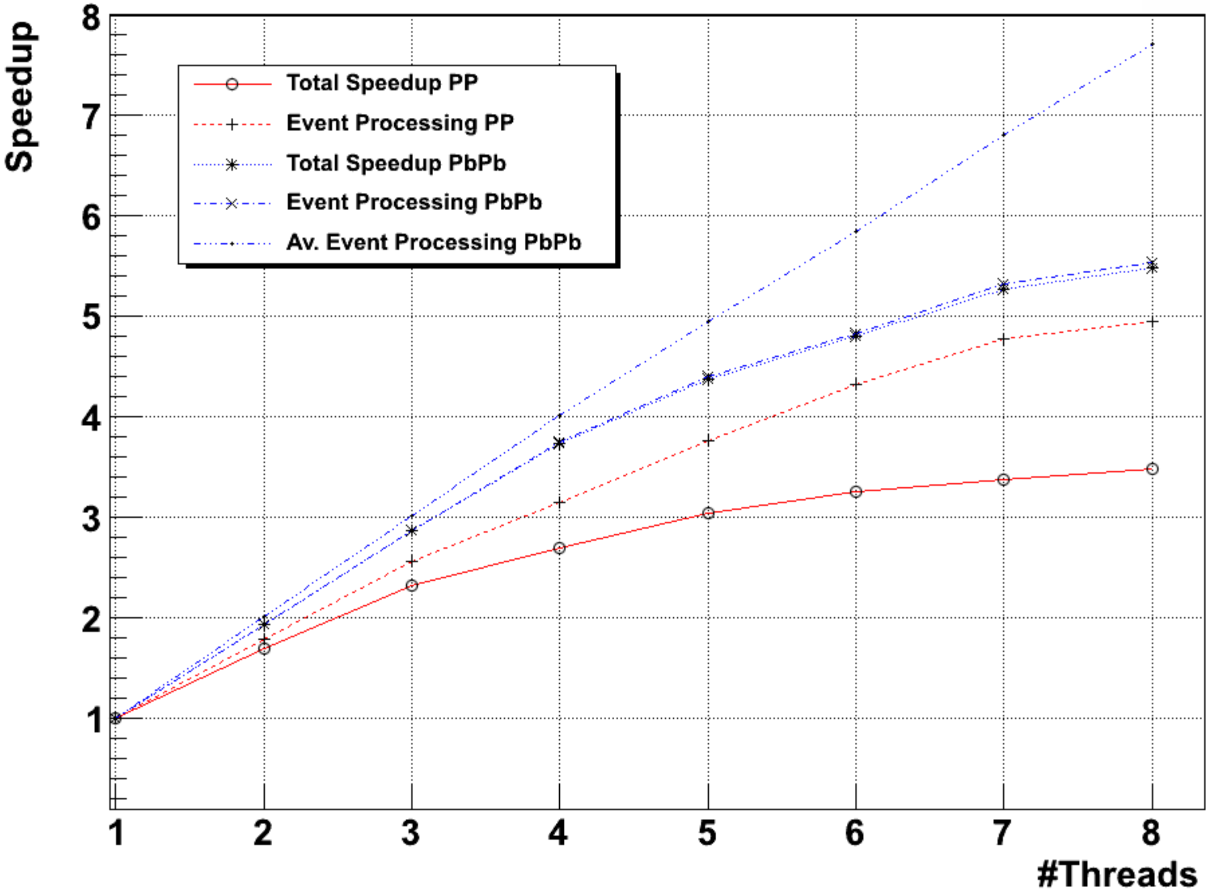
Testing Event Reconstruction

200 p-p events (red)
vs. 20 Pb-Pb events (blue)

ITS only, no QA

Max. Speedup of parallel processing only:
4.9 (p-p), 5.6 (Pb-Pb)

Test machine: 12 core Intel Westmere. 2.6 GHz, 12 MB LL cache.

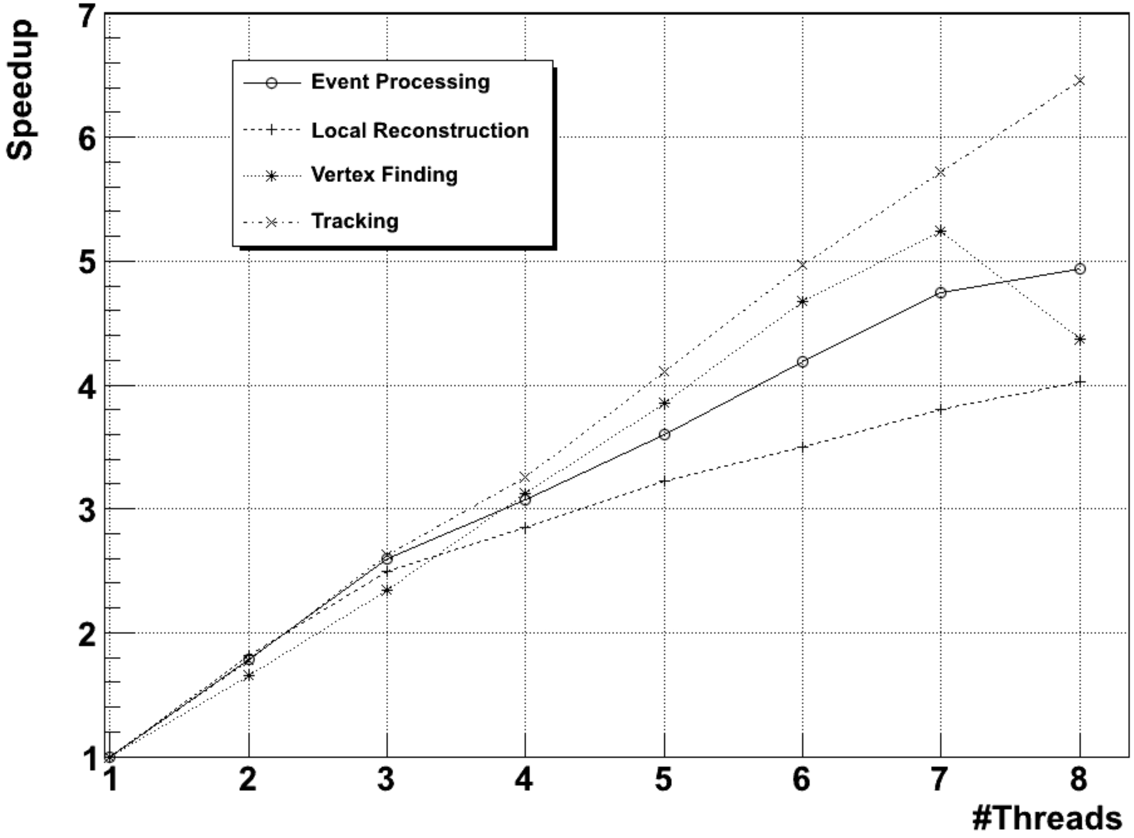


Synchronization avoids scalability. Led-Led processing is more computing intensive and uses less synchronization. Hence it scales better.

Testing Event Reconstruction

Performance characteristic:

1. 3,95 M calles of _tls_get_addr per pp-event. 1.45% of runtime.
2. **350k mutexes acquired per event, thereof around 90k Tstorage::Alloc***
3. Around 360k times malloc/free per event, 1.16% of runtime.



- The parallel initialization is locked and affects performance.
- Local reconstruction is the slowest task, due to intense IO.

Testing Event Reconstruction

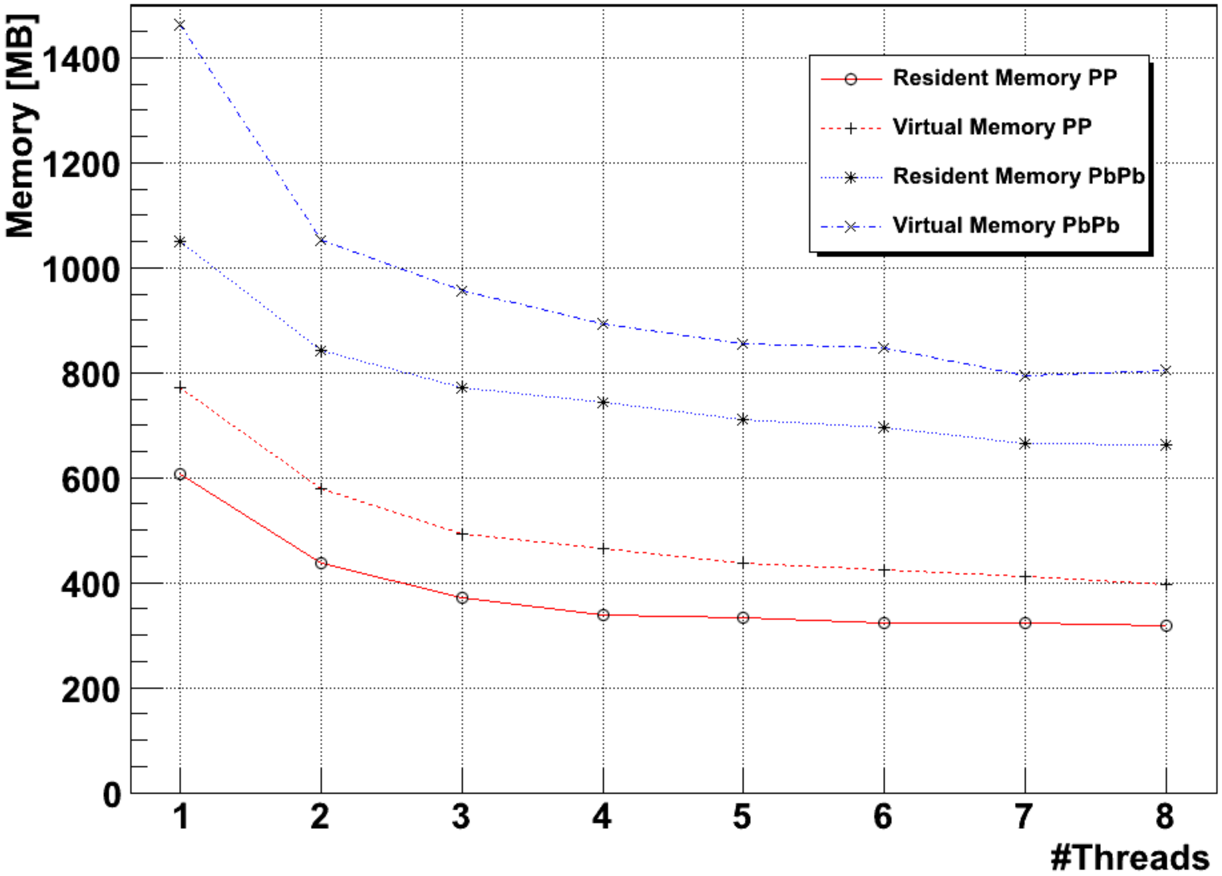
200 p-p events
vs. 20 led-led events

ITS only, no QA

Memory per thread:
400 MB (pp), 800 MB (PbPb)

Only Cint & TROOT shared.

Test machine: 12 core Intel Westmere. 2.6 GHz, 12 MB LL cache.



Each thread has an opened set of files for reading raw data and writing down results. For a Led-Led run these consume 200 MB and for a p-p run 133 MB virtual memory.

Conclusions

Transformation:

1. Introducing multi-threading *without expert knowledge* is possible.
2. Rewriting with the current tool is sophisticated and error-prone. It can be ***improved by using a new Clang*** front-end.

Performance:

1. Speedup of a Led-Led run rose to ***5.6 with 8 threads***.
2. Memory consumption is reduced by around 100 MB. Led-led reconstruction ***footprint still rises to 600 MB*** virtual memory per thread.
3. More memory can be shared, e.g. Geometry and CDB data.
4. Prototype can be used to evaluate performance for first design

Correctness:

1. With helgrind and MWPT, practical concerns can be evaluated.
2. All concerns of correctness have been addressed by provided tool.
3. Since validation and the corresponding tools are hard to apply, many items are not jet fixed appropriate, leading to instability.

Questions?