Support for User-Defined Metrics in the On-line Performance Analysis Tool G-PM

Roland Wismüller

LRR-TUM, Technische Universität München
Munich, Germany

Background

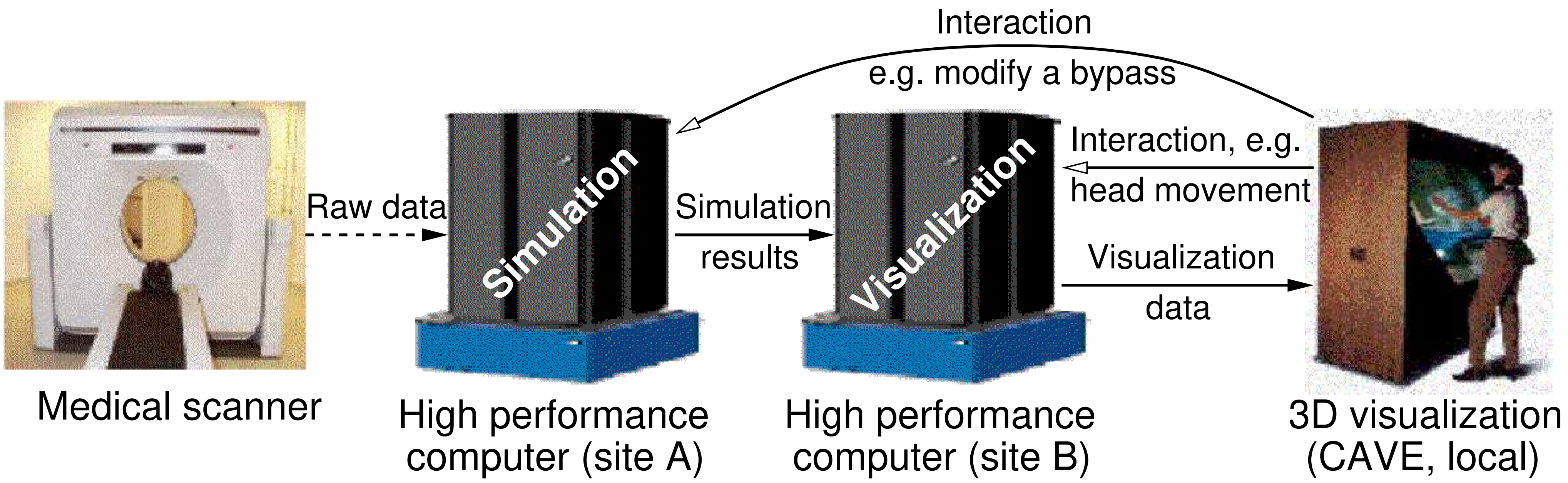
EU IST-project CrossGrid

- Aim: make the Grid usable for *interactive* applications
  - four parallel / distributed applications
  - tools (test, benchmarks, performance analysis / prediction)
  - middleware (portals, scheduling, monitoring, file access)
  - testbed
- Task 2.4: performance analysis tool for Grid applications
  - ICS-AGH, Cracow (Marian Bubak, Wlodzimierz Funika and others) and LRR-TUM

EU working group APART

- Automatic performance analysis

Motivation: Medical Simulation in CrossGrid



- Interactive simulation and visualization
  - response times are critical
- Requirements on performance analysis tools:
  - measurement and breakdown of response times
  - performance numbers for individual interactions

Goals of G-PM

- **On-line analysis and visualization**
  - direct correlation between performance and interactions
  - prerequisite for application steering
- **Support for user-defined metrics**
  - e.g. response time, convergence rate, data per interaction
  - with minimal effort for the user (application programmer)
- **Meaningful performance measurement in the Grid**
  - relate application performance to performance of Grid infrastructure
- **Efficient, distributed data evaluation**
  - scalability, perturbation of application
  - no centralized processing of event traces

Basic Approach of G-PM

- **G-PM tool offers a set of standard metrics**
  - related to applications, e.g. time and volume of communication and I/O, CPU and memory usage, ...
    - automatic instrumentation based on counters and timers, transparent to the user
  - related to Grid infrastructure, e.g. load of nodes, ...
- **User-defined metrics and probes**
  - probes: minimal manual source code instrumentation
    - important events, plus application data (optional)
  - user can define new metrics at run-time
    - basis: existing metrics and information from probes
    - flexibility, re-use of metrics and probes

Example: A Manual Measurement

- User wants to measure the amount of disk I/O for each interaction of a certain type
- Assume that there already is a counter for the total I/O volume
- Then, the user normally would instrument the code like this:

```
old = IO_volume();
// processing of interaction starts here
    .
    .
    .
// processing of interaction ends here
vol = IO_volume() - old;
MPI_Reduce(&vol, &total, ..., SUM, 0, ...);
if (rank == 0)
   printf("%d. interaction: %f\n",i,total);
```

Example: A Measurement with G-PM

- Idea of G-PM: only the relevant places are marked in the source code via probes:

```
begin(i);   // i indentifies the interaction
// processing of interaction starts here
    .
    .
    .
// processing of interaction ends here
end(i);
```

- The metrics to be measured then can be specified at run-time
  - based on the PMSL performance metrics specification language
  - measurement is then implemented by G-PM
    - including optimizations and distributed evaluation

PMSL Specification of the Example Metrics

```
IO_volume_for_interaction_vt(Process[] processes, File[] partners,
                              VirtualTime vt)
{
  PROBE end(Process p, VirtualTime vt);
  PROBE begin(Process p, VirtualTime vt);
  Process p;
  Value[] volume;
  volume[p] = IO_volume(p, partners, [START, NOW]) AT end(p, vt)
             – IO_volume(p, partners, [START, NOW]) AT begin(p, vt);
  return SUM(volume[p] WHERE p IN processes);
}
```

- Purely declarative language with single assignment form
- AT operator for dealing with probe events
- „Associative arrays“, set aggregation

PMSL Specification of the Example Metrics ...

- If the event rate is high, we may want to aggregate the values over small intervals of time
- Since each measurement value has a time stamp, the following specification does the job:

```
IO_volume_for_interaction(Process[] processes, File[] partners,
                          TimeInterval timeInt)
{
  VirtualTime vt;
  Value[] volume;
  volume[vt] = IO_volume_for_interaction_vt(processes, partners, vt);
  return SUM(volume[vt] WHERE volume[vt].time IN timeInt);
}
```

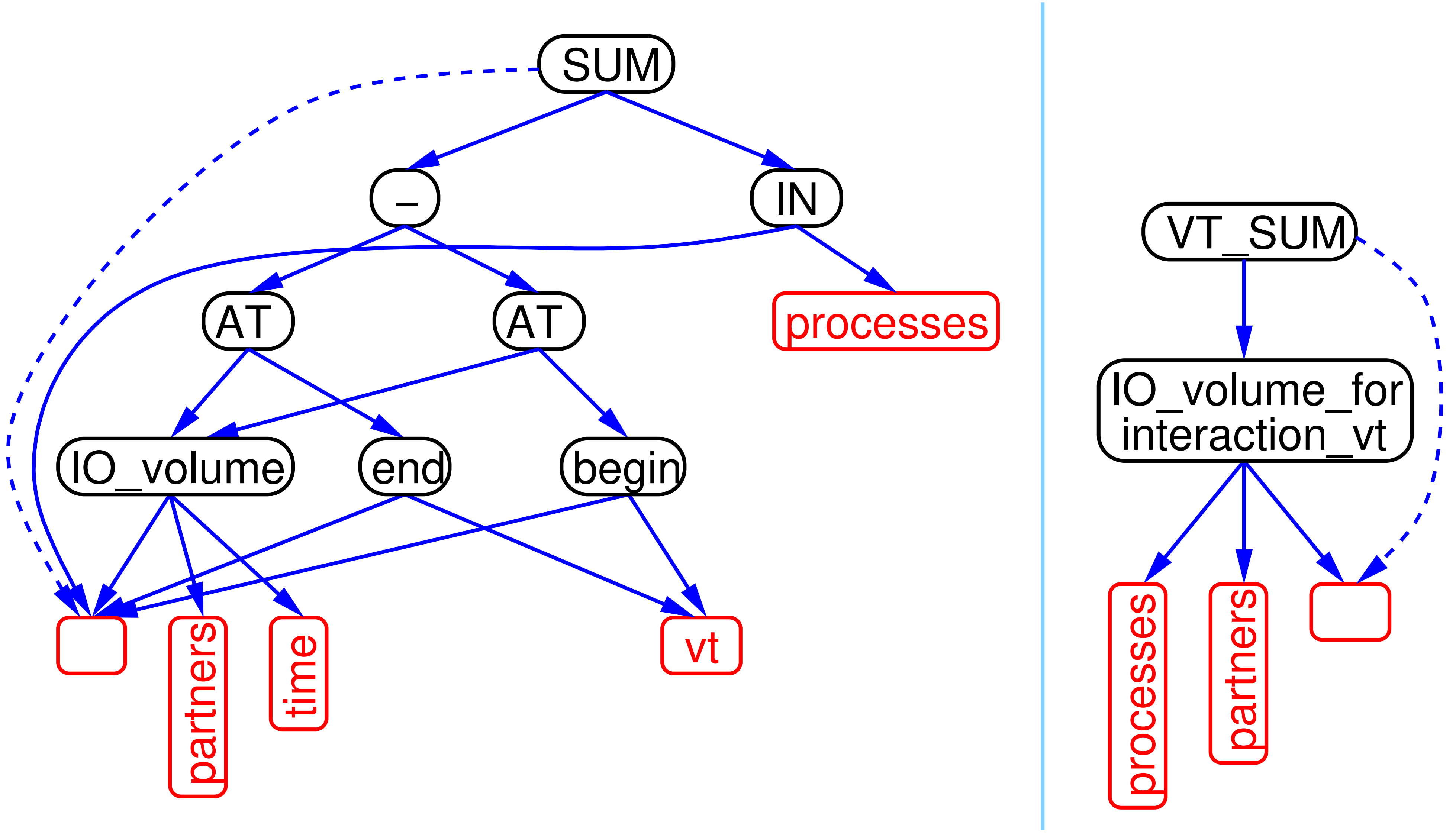
Implementation of Measurements (1)

Specification is translated into distributed dataflow graph

- When metrics is specified:
  - **translate** it into intermediate representation (IR): a DAG
  - parameters (objects, restrictions) still unknown
- When measurement is defined:
  - assign parameter values, **partial evaluation of IR**
    - if possible, „inline“ IR of base metrics
  - **optimize the IR**, e.g. push aggregations in time towards leaves
  - **transform IR into dataflow graph**, request inferior measurements and probe monitoring
  - **distribute the dataflow graph**, start the measurement

Implementation of Measurements (2)

IR of the two example metrics



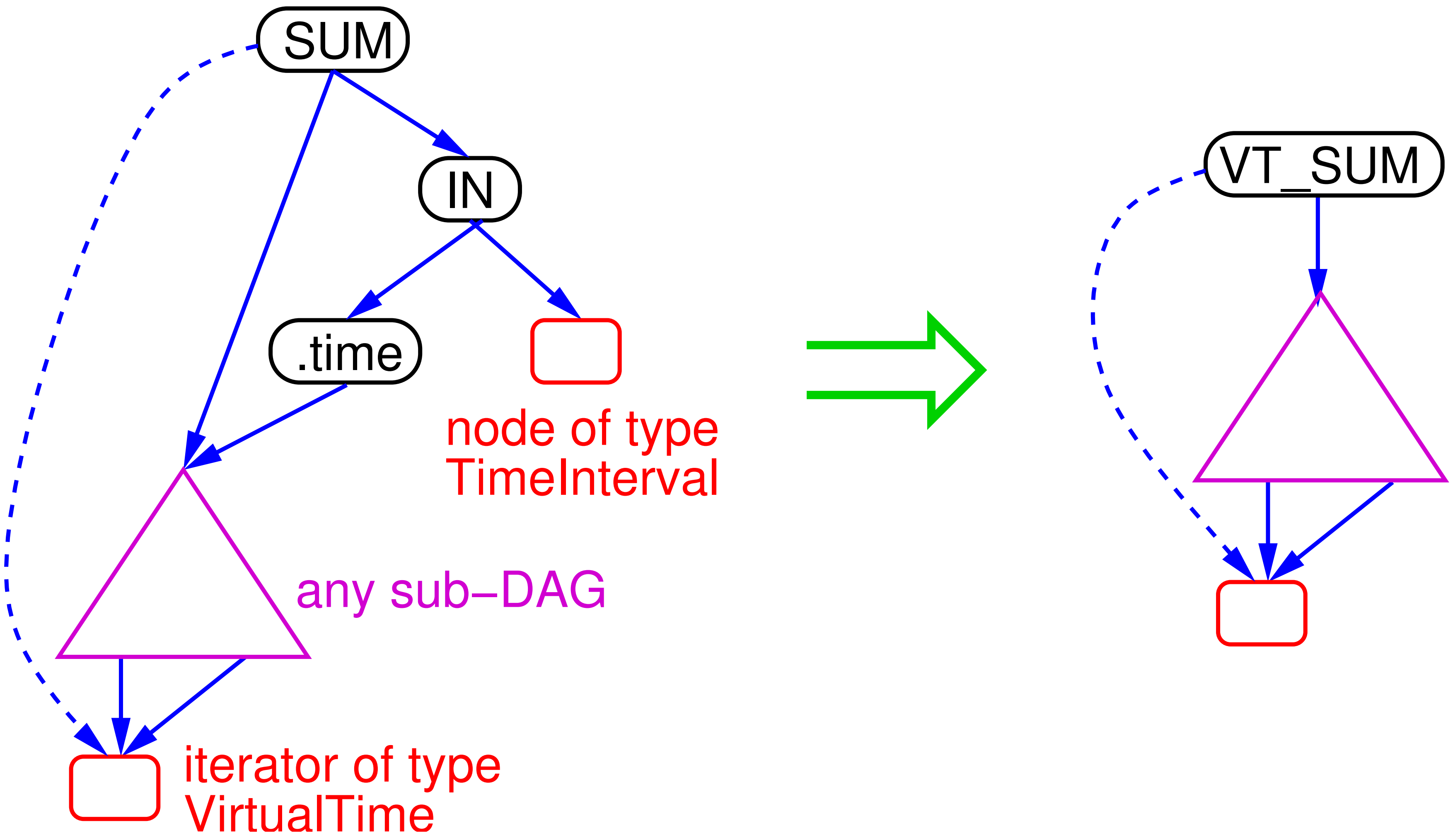
Implementation of Measurements (2)

```
IO_volume_for_interaction_vt(Process[] processes, File[] partners,
                                              VirtualTime vt)
{
  volume[p] = IO_volume(p, partners, [START, NOW]) AT end(p, vt)
                – IO_volume(p, partners, [START, NOW]) AT begin(p, vt);
  return SUM(volume[p] WHERE p IN processes);
}

IO_volume_for_interaction(Process[] processes, File[] partners,
                                         TimeInterval timeInt)
{
  volume[vt] = IO_volume_for_interaction_vt(processes, partners, vt);
  return SUM(volume[vt] WHERE volume[vt].time IN timeInt);
}
```

Implementation of Measurements (2)

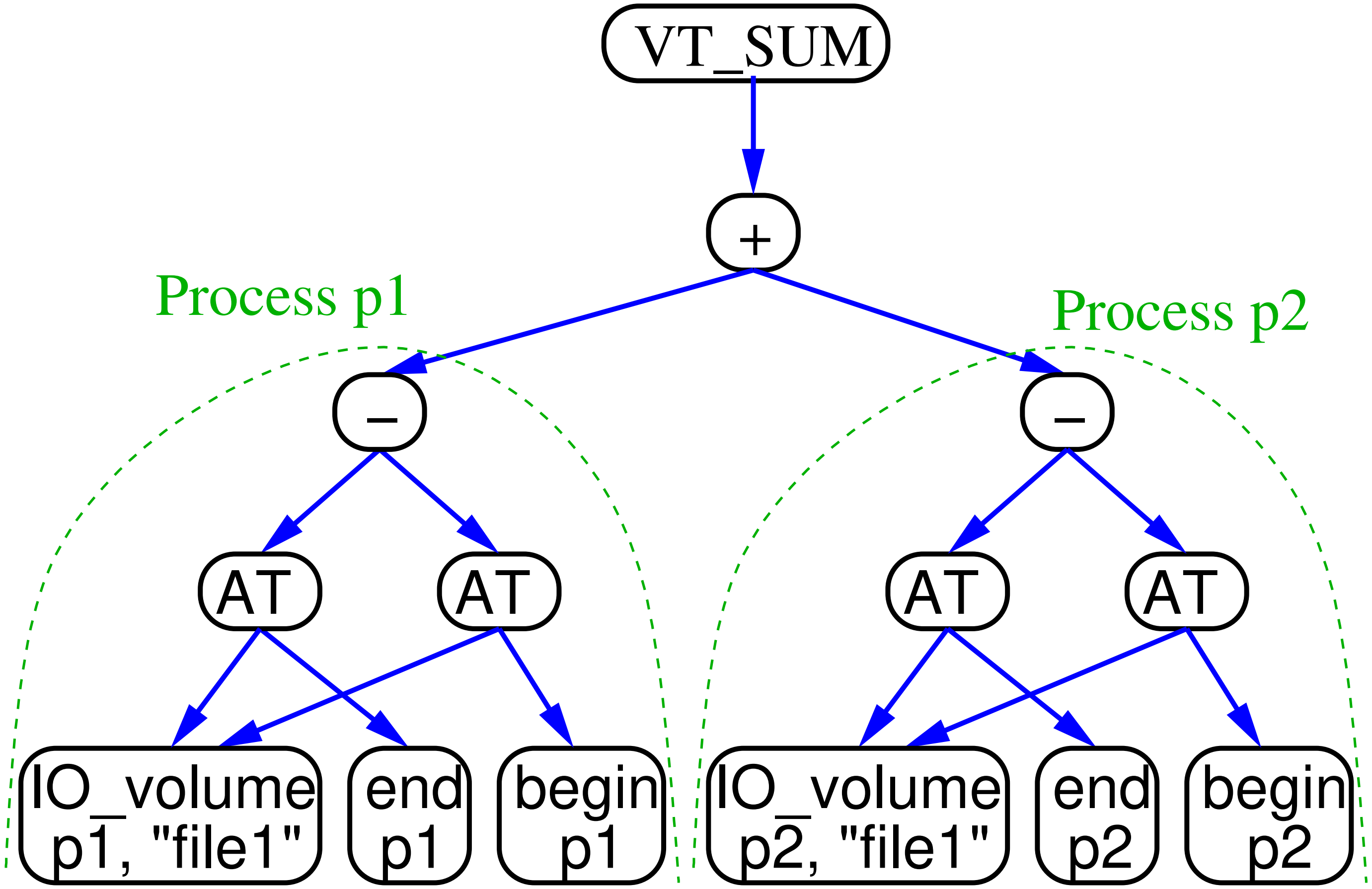
Graph pattern matching for aggregations in time



Implementation of Measurements (3)

Partially evaluated IR

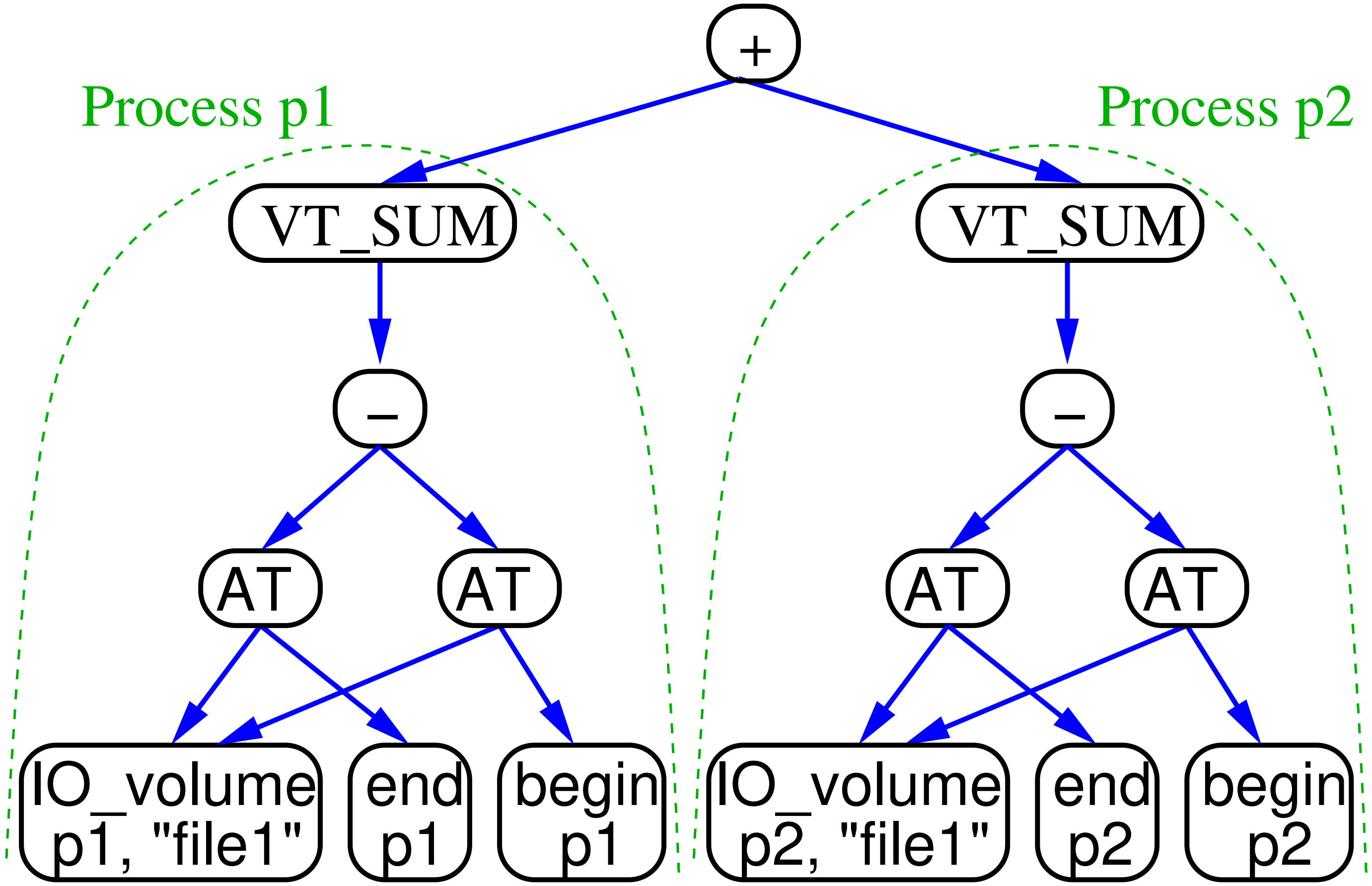
- Measurement for processes **p1**, **p2**; restricted to **file1**



Implementation of Measurements (3)

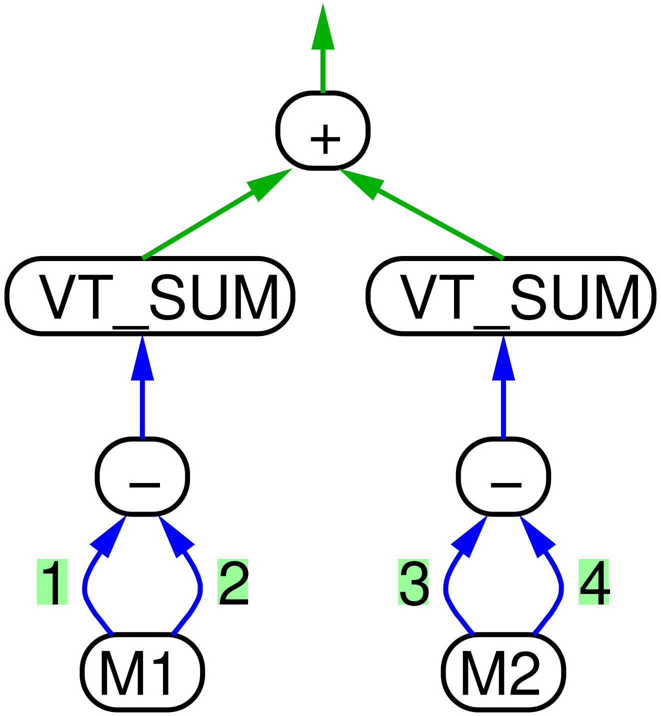
Partially evaluated IR after optimization

- Interchange + and **VT_SUM**: less data transfer



Implementation of Measurements (3)

Dataflow graph and evaluation



Callback	**Actions performed**
p_i : begin	get vt from event info request result of measurement i callback data = (vt, i==1 ? 2 : 4)
p_i : end	get vt from event info request result of measurement i callback data = (vt, i==1 ? 1 : 3)
measure–ment$_i$	pass (value,time,vt,requesterID) to node M_i

Related Work

Customizable trace analysis

- Pablo (Illinois): user-defined analysis / visualization by composing modules
- Paraver (Barcelona): menu-driven specification of metrics
- EARL (Jülich): language for trace analysis

Metrics specification languages

- MDL (Wisconsin): low-level implementation
- ASL (APART WG): specification of performance properties
- JavaPSL (Vienna): ASL implemented via Java classes

Status and Future Work

Status:

- PMSL and its translation are fully implemented in 2nd prototype of G-PM
  - demonstrated at this meeting
- However, evaluation of measurements is currently done in a centralized way

Future work:

- distributed evaluation of dataflow graphs in monitoring system OCM-G
- DAG transformations for optimizations
- extensions of PMSL