CODE INSTRUMENTATION AND OPTIMIZATION - A Static - Dynamic Framework

Alexandra Jimborean, Matthieu Herrmann, Vincent Loechner, Philippe Clauss and Eric Violard
INRIA, LSIIT, Université de Strasbourg – Pole API – Bd. S. Brant – F-67412 ILLKIRCH Cedex

1. Introduction

Most programs are designed to be executed sequentially, but with the advent of multi-core processors, automatic parallelization became the holy grail of parallel computing.

2. Objective

Entirely static parallelization is not alway possible, as a consistent amount of information is known only at runtime. Therefore a static-dynamic framework is required.

3. Step by step

- Instrument the code to observe the memory accessing behavior of the program
- Analyze dependencies
- Speculative parallelization
- Check correctness
- Roll back if incorrect

4. Instrumentation

- Statically inject code to collect info
- Dynamically process the acquired info

- Focus on loops: most time of the program execution is spent in loops
- Identify promising zones using pragmas = hints for the compiler

5. Examples

#pragma instrument_mem_add : mark the region to be instrumented; the compiler inserts code to determine the accessed memory locations

#pragma unroll_loop : causes the compiler to unroll the loop; may take some parameters

```
# pragma instrument_mem_add
{
   for(int i=0; i<N; i++)
      a[i] = 2 * i;
}
```

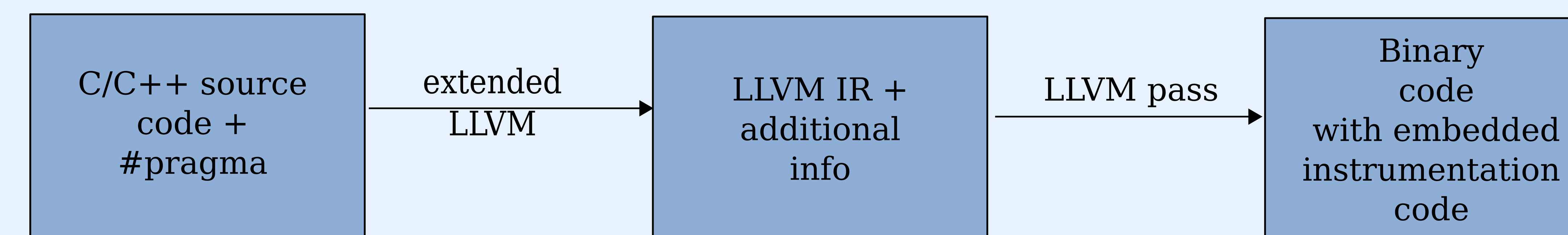
Pragma inserted in the source-code

6. Static component of the framework: LLVM

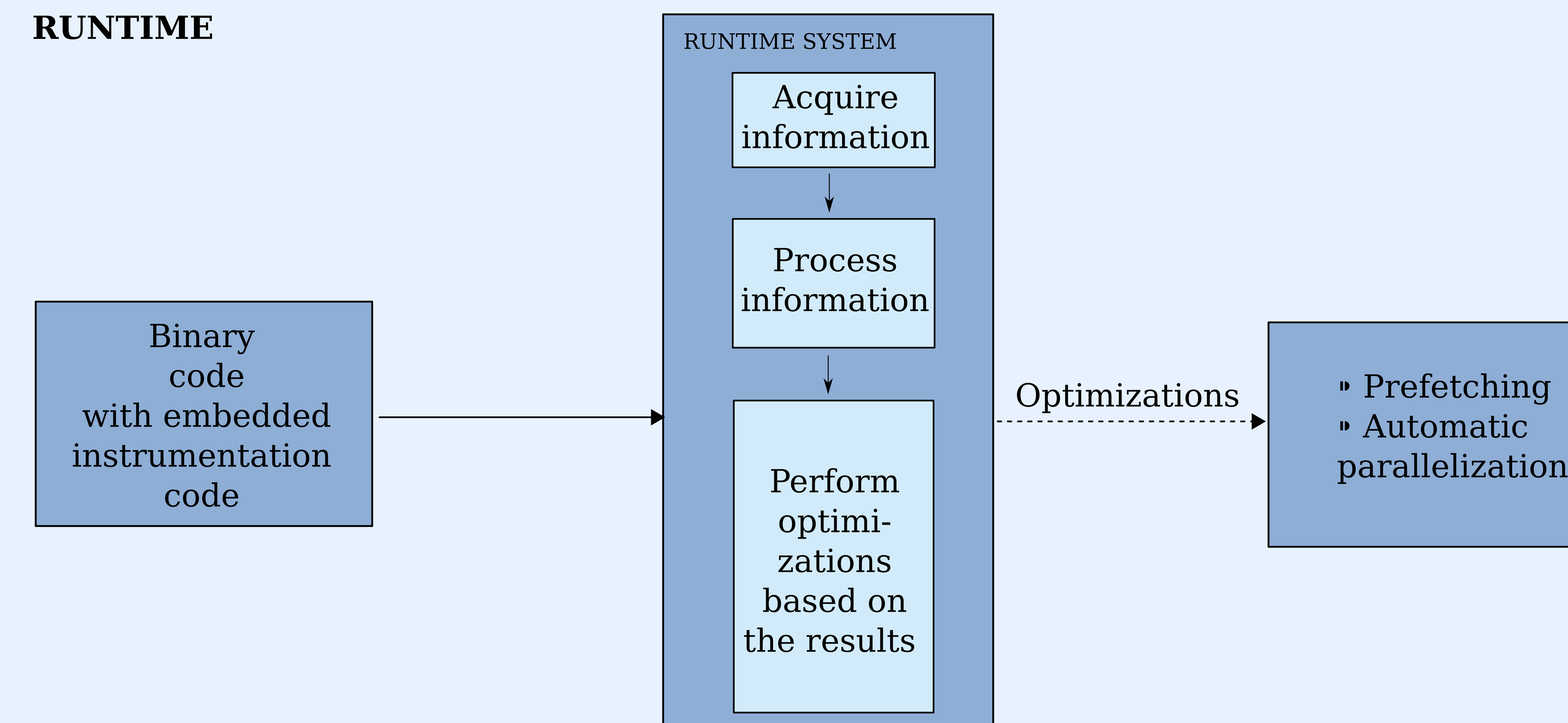
The LLVM compiler and the Clang front-end have been extended to generate binary code following the pragma semantics.

In addition to the original code, the compiler generates instrumented and optimized versions or provides the information available statically, required to generate these versions at runtime.

COMPILE TIME



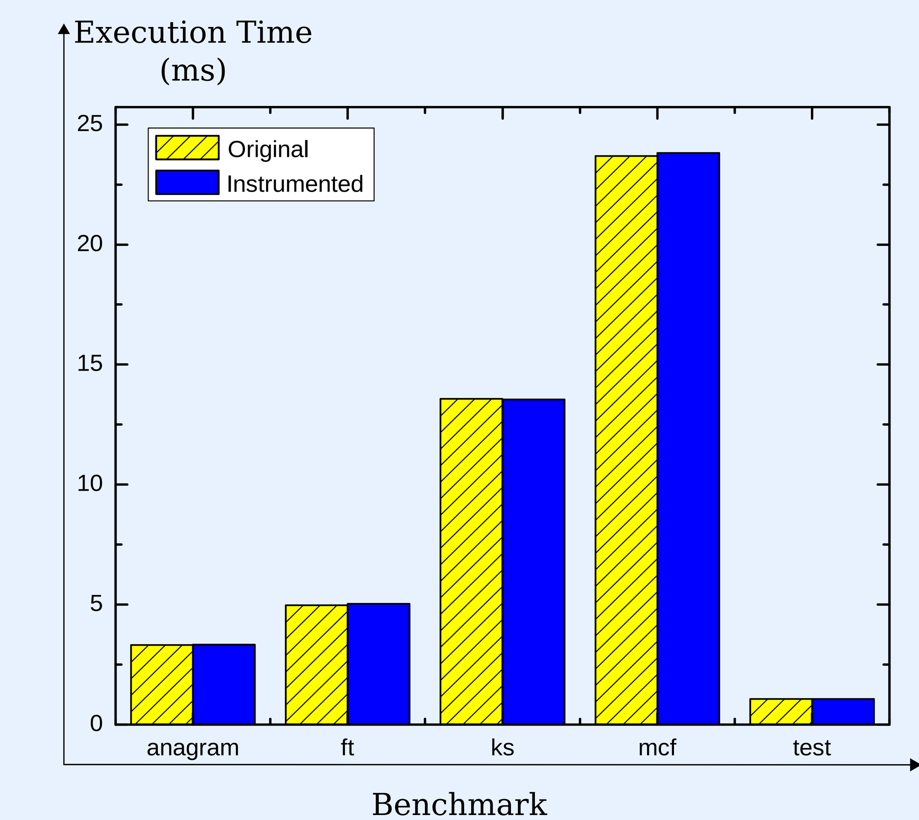
RUNTIME



7. Dynamic component of the framework: Runtime system

The role of the runtime system (RS) is to decide the execution flow of the program by patching the binary code. It selects among the versions generated statically and processes the information acquired at runtime. The RS takes as input the binary code generated by the compiler as well as a set of parameters computed statically.

8. Instrumentation overhead tends to 0



9. Results

Experiments conducted on problems from the Pointer Intensive Benchmark Suite (anagram, ft, ks) , SPEC CPU 2006 (mcf) and a test problem reveal that the slowdown induced by the instrumentation code is very limited, with values from -0.13% to 1.26%.

10. Conclusions

This study is aimed to describe the memory accessing behavior of the programs.

The limited overhead is the consequence of instrumenting only a sample of the iterations in the analyzed loops. Sometimes, a small speed-up is obtained as a side-effect of some code modifications.

The slowdown induced by generating new code or performing additional steps at runtime has to be hidden by optimization techniques.
These are achieved based on the results obtained from instrumentation.

11. Future steps

- perform dynamic prefetching
- parallelize automatically and dynamically loop-intensive codes