The source code infrastructure SageII applied to thread extraction

INRIA/IRISA

DESY / Zeuthen

Thuesday, October 12, 2004

Introduction

Source to source code transformations and analysis are key issues in many code generation and optimization softwares.

- Compilation and preprocessing
- Parallelizing transformations
- Code instrumentations

This talk is about:

- An original case of study: advance thread extraction mechanism in the design of embedded systems
- The case of study framework: SageII, a C analysis and transformation infrastructure

The thread extraction

Aim of the study: extract potential sections of C code that can be executed on a remote device: threads

We propose a speculative thread extraction mechanism

- What for: Part of the design process of embedded system design:
  - Specialized co-processors or units in S.O.C.
  - Multi-core processors.
- Study restrictions: target independent
  - analysis restricted to the high level code
  - no shared memory available
- We Perform a Source to Source transformation : C code -> C code + thread code

The Speculative Thread Execution

Dynamic approach

Extractions based on program execution information: profiling

- Program execution traces
  - Generate program execution paths
  - Function independent: not limited by the function organization
- Memory access traces
  - Control indirect memory access: compute the memory mapping

Imply a speculative execution to keep control over the effective execution of the thread:

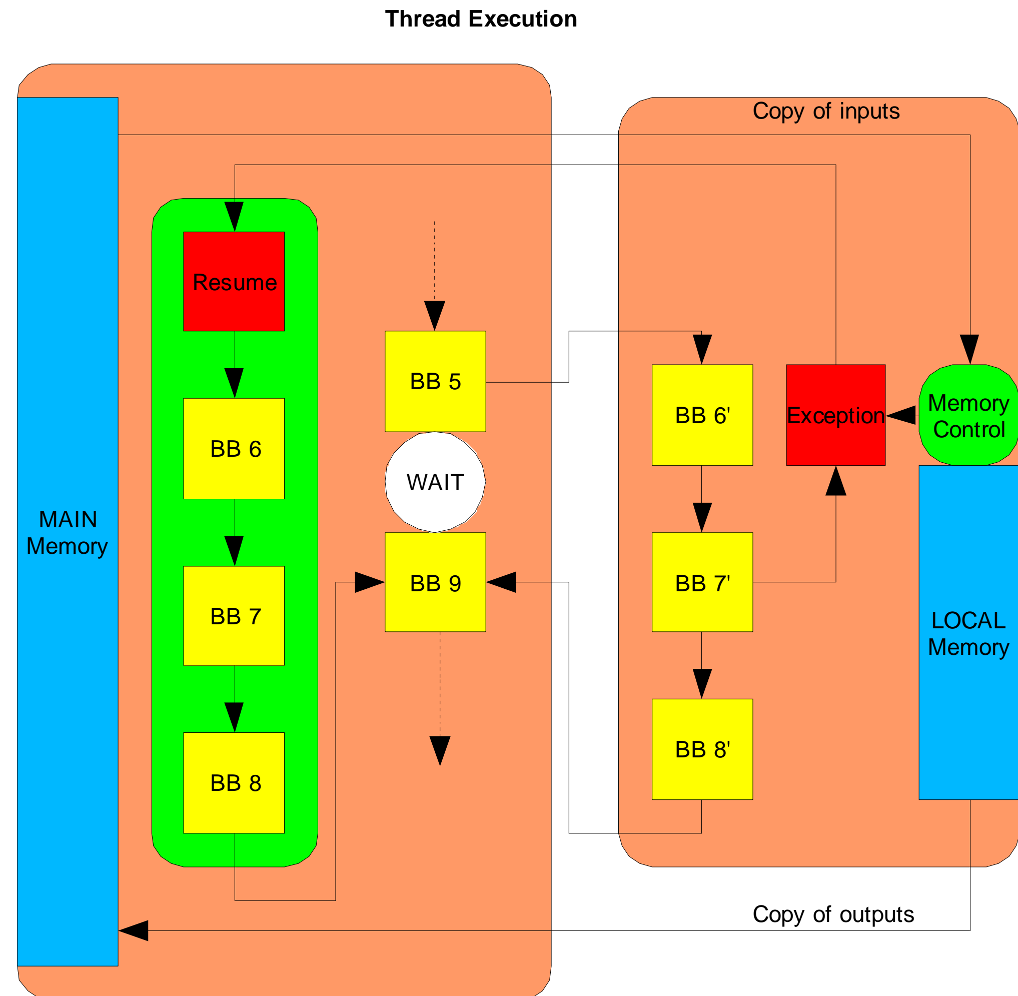
- Path taken by the execution must follow the trace
- Memory accesses must be restricted to the thread scope

The set of executions is considered representative

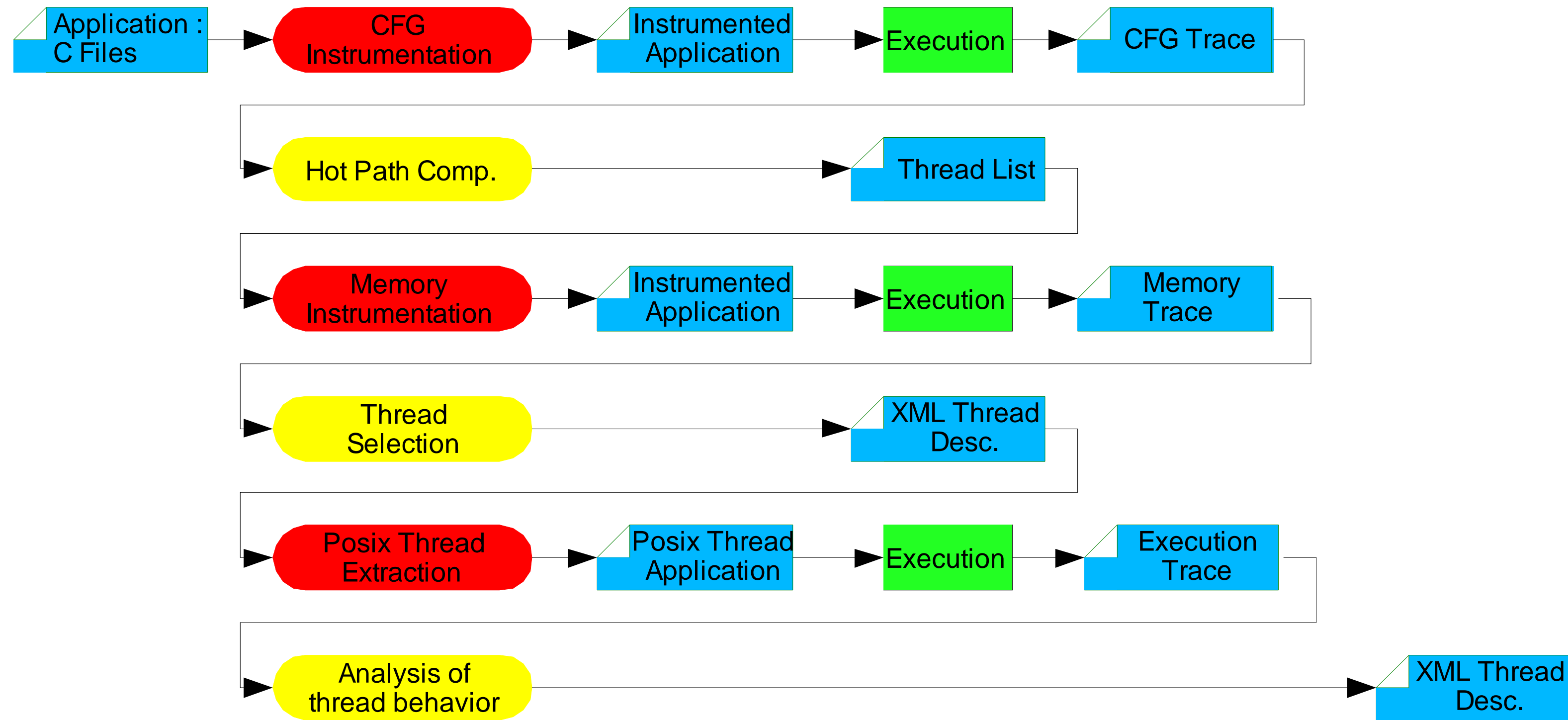
The Thread execution

Speculative execution of a thread:

1. Copy of memory inputs
2. Remote execution & wait
3. If exception
   - Resume execution
   - Execute original code
4. Else
   - Copy of memory output
5. Continue



A Trace Based Approach

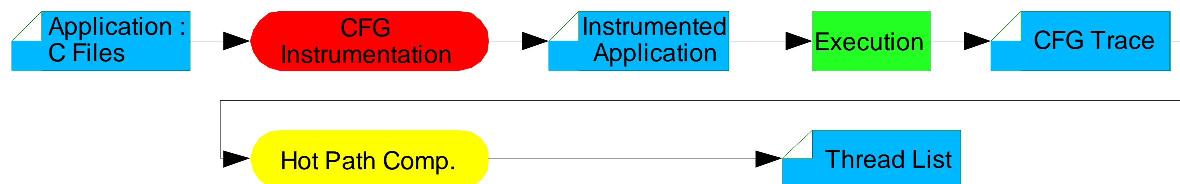


At each level of the extraction process we will consider the infrastructure requirements

Hot path Analysis

Computes repeated sequences of Basic Blocs

- Thread extraction requirements:
  - A control flow graph (CFG) abstraction of the C code
  - A source code instrumentation of basic blocks
- Infrastructure requirements:
  - Abstraction of the program control flow instructions: statements
  - List of C operations contained in basic blocks
  - Methods to build and insert function calls: to perform the instrumentation
  - Unparsing of the statement abstraction



Memory traffic Analysis

Computes the size and the source of memory segments handled by the thread

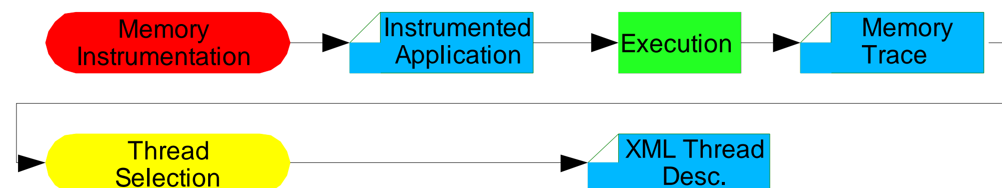
- Use a **static analysis** for scalars and a **dynamic analysis** for arrays and pointers

– Thread extraction requirements:

- An abstraction of variables containing their size and type
- Dependences between operations on variables: def/use chains list
- Memory operations instrumentation: detection of pointers and arrays accesses

– Infrastructure requirements:

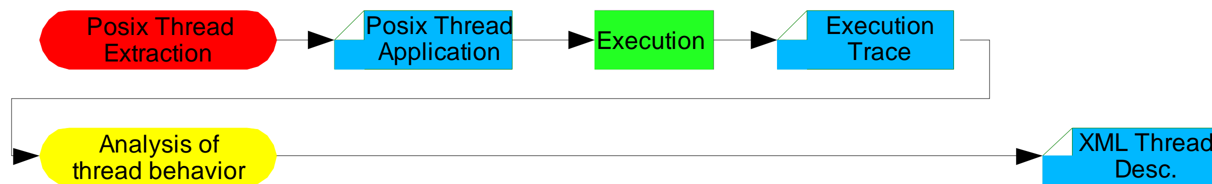
- A type abstraction: defining the nature and the size of symbol's types
- A Symbols abstraction: a symbol table including the scope and the type
- A precise C expression abstraction
- Methods to build and insert function calls inside the expression abstraction
- Unparsing of the expression abstraction



Posix Thread Extraction

Transform the original code into a threaded C code

- Thread extraction requirements:
  - Generation and insertion of declaration instructions: function prototypes and local symbols
  - Thread code generation including
    - memory synchronizations mechanisms
    - miss-predicted execution paths exception calls
    - miss-predicted memory access exception calls
- Infrastructure requirements:
  - An abstraction of the whole syntax tree of the program (AST)
  - Methods for typed symbol creations and declarations in a given scope
  - Methods for adding, deleting or moving C code in the AST.



Source code infrastructures

Tools that build an abstraction of a source code

Composed of:

- An internal representation of the code abstraction
- A library of classes for the analysis and the manipulation of the abstraction.
- A library of classes for the parsing, scanning and unparsing of the code abstraction

Characterized by:

- The code abstraction level: description accuracy
- The semantical information computation: symbol scopes and type definitions
- The range of code accepted by the infrastructure: simple codes or a full support of ISO specifications.
- The unparsing quality: original code structure and indentation respect, ...
- The number and the quality of available tools based on it: def/use chains, ...

Mainly used for:

- the design and the experimentation of new compilation passes
- The cost reduction of the compiler design

Overview of existing infrastructures

The most famous: SUIF as Standford University Format

- a unique internal representation of programs for different languages
- manage only the abstract syntax tree level

The most advanced : PUMA as PUre MAnipulator

- accurate representation of C and C++ Programs
- computes symbols and types information

The most precise: SageII

- accurate internal representation of C programs with type control
- wide range of C codes supported

Stanford University Intermediate Format

SUIF is designed for cooperative research. Still in development

- The SUIF infrastructure is mainly used as an intermediate format:
  - based on an unique abstraction for all high level languages
  - many high and low level analysis available
- Two incompatible versions available:
  - The first is based on a simple AST and includes a compiler and an unparser
  - The second requires the use of a commercial binary front end: EDG C. No unparsing.
- Main issues of the first version:
  - the code abstraction is normalized: lost information
  - no semantic information computed: not type control or symbol management
  - partial C support: restricted to a few number of programs

Puma

Designed for the Aspect C++ language by the University of Magdeburg, Germany. Still in development

- The most advanced infrastructure:
  - Complete and precise abstraction of both C and C++
  - Proposes three differents abstractions:
    - An AST for the C preprocessor
    - An abstraction of the textual form of the code: a list of tokens
    - An accurate AST for C and C++ code
  - Includes a symbol and type definition computation
- It incorporates various tools
  - Multi-files multi-directories projects management
  - Token sequence and subtree pattern matching

SAGE II

SageII is a new C code analysis and transformation infrastructure based on Sage++

Sage++: first original high level language framework, from University of Indiana, 1994

- Precise internal abstraction of source code by an AST (Fortran initially)
- Programmable unparsing of the AST to the source code
- Symbol table with scope management

Sage II adds:

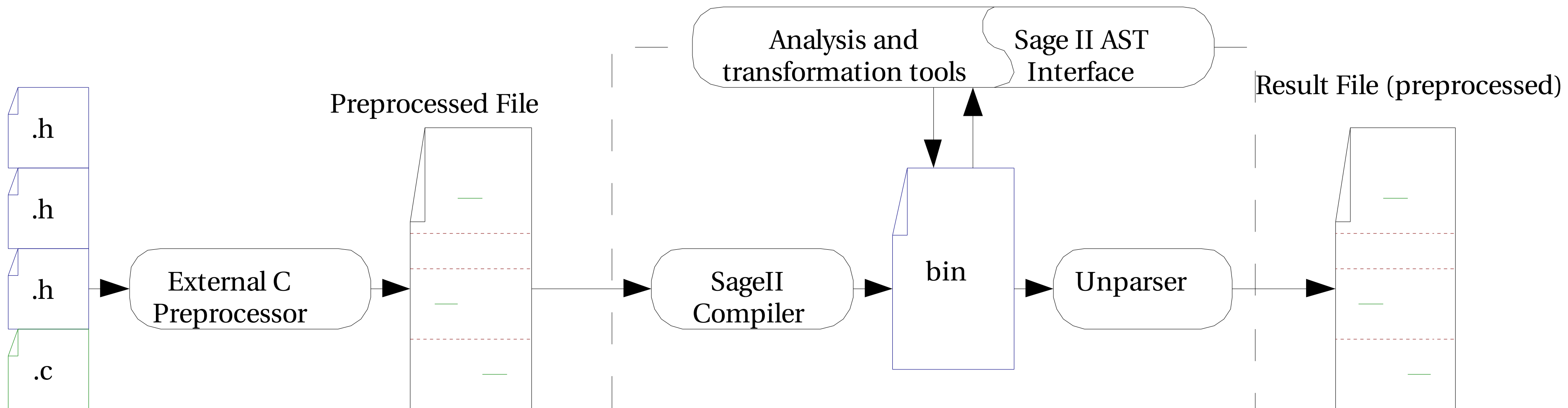
- A precise internal abstraction of the C language
- An abstraction of all types with a type control management
- a full support of ISO IEC C99 standard
- a configuration system to support C extensions : GCC
- an automatic AST scanner

Successfully tested over more than 1 million lines of code:

- tested on standard and system headers (stdio.h, Xlib, ..) on Linux, Solaris, MacOs;
- tested and verified on gcc, gzip, h263, pgp, momusys, ghostcript, C torture test, ...

SAGE II work flow integration

The infrastructure builds a binary representation of C program abstractions



SAGE II: Implementation of Thread extraction

Sage II is used to perform each level of the thread extraction process:

- Hot path Analysis:
  - Computes the source level Control Flow Graph and link it to the AST: basic block <-> C instructions
- Code instrumentations and Posix thread generation
  - Creation of any arbitrary C objects:
    - » Symbols
    - » Automatic declaration of symbols in the right scope (functions and variables)
    - » Modification and Generation of all kind of C expressions: function calls, operations, ...
  - An exact unparsing of the Internal abstraction: identical to the original preprocessed code
- Memory access analysis
  - Def/use chains of all variables: symbol dependences
  - Type and size of variables: precise management of pointers and arrays.

SAGE II: current state

SageII infrastrucure based tools:

- Induction variables computation
- Restrict pointers dynamic analysis
- Compiler front-end
- Common transformations: loop unrolling, function inlining, ...
- XML description generation of C programs
- ...

Improvement:

- Preprocessor management
- Memory management system (recycling of unused nodes),

Future works:

- extension to C++

Conclusion

Sage II: a powerfull infrastructure to develop complex analysis and transformations based on:

- The construction of the C program Abstract syntax tree
- A full symbol and a type control
- A set of utilities and a C++ interface

Effectiveness of Sage II on a concrete case of study: a speculative thread extraction mechanism

– A dynamic and fully automatic extraction:

- Potential thread detection
- Memory dependences analysis
- Thread-based program transformation

Small Example

Original Program

```
int main(int argc, const char *argv[])
{
 int index=0;
 unsigned long crc=0;
 const char string[30];
 unsigned int stringLength;

 if (argc < 3) {
  printf("usage: %s <string> <number>\n", argv[0]);
  exit (-1);
 }

 if (argv[1]) strncpy(string,argv[1], 30);

 for (stringLength=0; string[stringLength]; stringLength++);

 for (index = 0; index < 100; index++)
  crc ^= string[index % stringLength];

 index = stringLength = crc + 1;
 printf("Crc=%i\n", crc);

 return crc + 1;
}
```

Static array detected:
no memory instrumentation needed

Hot Path results

```
<?xml version="1.0"?>
<WorkingSet cycles="1156" static="13" dyna="0" >
        <WS number="10" Pos="21" End="222"
                freq="500" exec_ratio="0.865052" cycles="2" >
                <node cfg="1" bb="12"  />
                <node cfg="1" bb="13"  />
        </WS>
</WorkingSet>
```

Detection of an hotspot for Basic Blocks 12 and 13

Small Example: the thread description

```
<?xml version="1.0" encoding="ISO-8859-15" ?>
<!DOCTYPE ThreadExtraction SYSTEM "thrdEx.dtd">
<ThreadExtraction xmlns:thrdEx="thrdEx.dtd"  date="19:19:23, Friday 01 October 2004"
                project="smallApplication.ref.bic" targetFile="smallApplication.ref.bic" >
<thread number="10" function="main" functionId="2" >
<limits type="statements" >
        <statement id="17" />
        <statement id="18" />
</limits >
<communications  >
        <input type="symbol" >
                <symbol type="define" name="index" id="5" />
                <symbol type="scalar" name="crc" id="6" />
                <symbol type="scalar" name="stringLength" id="8" />
                <symbol type="array" name="string" id="7" size="30" />
        </input >
        <output type="symbol" >
                <symbol type="scalar" name="crc" id="6" />
        </output >
</communications >
</thread >
</ThreadExtraction >
```

C operations computed inside the trace

C declaration only : not read inside the thread

Speculative Size of the array

Variables written in the thread and read outside

Small Example: the thread call

```
(...)
 {
 int  size_out_arg_thread_10 = sizeof (crc) + sizeof (stringLength) + sizeof (string);
 char  *out_arg_thread_10 = (char *) (malloc(size_out_arg_thread_10));
 int  size_in_arg_thread_10 = size_out_arg_thread_10 + sizeof (size_out_arg_thread_10);
 char  *in_arg_thread_10 = (char *) (malloc(size_in_arg_thread_10));
 memcpy(in_arg_thread_10, &crc, sizeof (crc));
 memcpy(in_arg_thread_10 + sizeof (crc), &stringLength, sizeof (stringLength));
 memcpy(in_arg_thread_10 + sizeof (crc) + sizeof (stringLength), &string, sizeof (string));
 memcpy(in_arg_thread_10 + sizeof (crc) + sizeof (stringLength) + sizeof (string), &out_arg_thread_10, sizeof (out_arg_thread_10));
 memcpy(in_arg_thread_10 + sizeof (crc) + sizeof (stringLength) + sizeof (string) + sizeof (out_arg_thread_10), &pad, pad);
 if (pthread_create(&pthread_10, NULL, thread_10, in_arg_thread_10))
   fprintf(stderr, "can't create thread_10\n"), exit(1);
 if (pthread_join(pthread_10, (void *) (&out_arg_thread_10)))
   fprintf(stderr, "can't join thread_10\n"), exit(1);
 if ((int ) (out_arg_thread_10) == 1)
   fprintf(stderr, "array bound check error , program exiting...\n"), exit(1);
 if ((int ) (out_arg_thread_10) == 2)
   fprintf(stderr, "pointer deref bound check error , program exiting...\n"), exit(1);
 crc = *((unsigned long *) (out_arg_thread_10));
 stringLength = *((unsigned int *) (out_arg_thread_10 + sizeof (crc)));
 string = *((const char (*)[30]) (out_arg_thread_10 + sizeof (crc) + sizeof (stringLength)));
 }
(...)
```

Declaration of the communication memory

Copy of needed variables

Thread Call

Exections caught

Retreive needed variables

Small Example: generated thread code

```
(...)
void *thread_10(void *arg)
{
void  *start = arg;
int  index;
unsigned long  crc = *((unsigned long *) (arg));
unsigned int  stringLength = *((unsigned int *) (arg + sizeof (crc)));
const char  string[30] = *((const char (*)[30]) (arg + sizeof (crc) + sizeof (stringLength)));
char  *out_arg_thread_10 = *((char **) (arg + sizeof (crc) + sizeof (stringLength) + sizeof (string)));
void  *end = arg + sizeof (crc) + sizeof (stringLength) + sizeof (string) + sizeof (out_arg_thread_10);

for (index = 0 ; index < 100 ; index++)
  crc ^= *((const char *) (array_bound_check((void *) (string), index % stringLength * sizeof (const char  ), sizeof (string))));

memcpy(out_arg_thread_10, &crc, sizeof (crc));
memcpy(out_arg_thread_10 + sizeof (crc), &stringLength, sizeof (stringLength));
memcpy(out_arg_thread_10 + sizeof (crc) + sizeof (stringLength), &string, sizeof (string));
return (void *) (out_arg_thread_10);
}
(...)
```

Declaration and initialization of local variables

Execution with a control of the speculative memory

Copy of thread results