POLITECNICO DI TORINO
Repository ISTITUZIONALE

Dynamic Trace-Based Data Dependency Analysis for Parallelization of C Programs

Original
Dynamic Trace-Based Data Dependency Analysis for Parallelization of C Programs / Lazarescu, M.T., Lavagno, L.. - ELETTRONICO. - (2012), pp. 126-131. (IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM 2012) Riva del Garda, Italy September 2012) [10.1109/SCAM.2012.15].

Availability:
This version is available at: 11583/2507480 since: 2020-07-05T15:30:09Z

Publisher:
IEEE

Published
DOI:10.1109/SCAM.2012.15

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

16 October 2026

Dynamic trace-based data dependency analysis for parallelization of C programs

Mihai T. Lazarescu
Politecnico di Torino
mihai.lazarescu@polito.it

Luciano Lavagno
Politecnico di Torino
luciano.lavagno@polito.it

***Abstract*—Writing parallel code is traditionally considered a difficult task, even when it is tackled from the beginning of a project. In this paper, we demonstrate an innovative toolset that faces this challenge directly. It provides the software developers with profile data and directs them to possible top-level, pipeline-style parallelization opportunities for an arbitrary sequential C program. This approach is complementary to the methods based on static code analysis and automatic code rewriting and does not impose restrictions on the structure of the sequential code or the parallelization style, even though it is mostly aimed at coarse-grained task-level parallelization. The proposed toolset has been utilized to define parallel code organizations for a number of real-world representative applications and is based on and is provided as free source.**

***Index Terms*—C program parallelization, KPN, C-to-C rewrite, execution trace, data dependency analysis, graph analysis**

I. Introduction

There is an urgent need to parallelize massive amounts of legacy sequential code to better utilize processors and systems that refocus from the acceleration of the execution of a single-thread to the increase of the overall throughput, by means of multi-processor architectures. However, even when parallelism is taken into account from the start of a project, writing programs for efficient execution on parallel architectures is still considered a difficult task.

Automated software parallelization has been extensively explored at the instruction and loop levels, which are appropriate for VLIW and vector processors. By contrast, parallelization opportunities at the *task* level, which are best suited for modern multi-core processors, were less explored, with some notable exceptions [1], [2]. Most of the latter techniques are so far restricted to specific types of loops and data access patterns.

The innovative toolset, developed in the context of the European project HEAP, addresses these challenges. It helps software developers to profile and parallelize existing sequential C-language applications by exploiting top-level parallelism. It synergistically uses and extends past work of [3], [4], based on *runtime, full scope* data-dependency tracing and sophisticated graph visualization techniques, to allow code developers to *optimistically* find the best *manual parallelization opportunities*. The developer can use the toolset to quickly detect *possible* parallelization opportunities whose *effective suitability* for parallelization have to be subsequently checked by other means, e.g., by code inspection.

The approach proposed in this paper supports any style of parallel code writing, including, but not limited to, the intrinsically race-free Kahn Process Network style [5].

Section II of the paper overviews other approaches and tools for sequential code parallelization. Section III presents the toolset flow and the operation of component tools. Section IV demonstrates the toolset use for analysis and parallelization of a real-life application and Section V concludes the paper.

II. Related work

Code parallelization is one of the most widely studied topics in compilers for parallel machines since the 1970's. Most of previous work has focused on selection of code segments within innermost loops (`do` in Fortran, `for` and `while` in C) that can be executed either fully in parallel (`do-all`) due to the lack of dependencies, or as a software pipeline [6]–[8].

These techniques are efficient for applications in specific problem domains (physics, fluid dynamics, structural engineering), but quite limited in the general case and cannot fully exploit architectures developed for gaming and multi-media applications in the PC world.

Thus, there is a strong need for techniques that assist the developer to manually partition an application beyond the limitations of automated analysis, for instance at top program level (as opposed to the innermost loop level) [9], [10].

A technique similar to the one implemented by our toolset was proposed in [3]. Our toolset extends this approach by providing techniques based on data compression and advanced visualization to effectively display the very large amount of data generated by data dependency tracing for, e.g., a large video encoding or decoding application.

Several compilation and debugging tools, often based on proprietary extensions of the C language, have also been proposed by dominant industrial players. For example, Apple introduced recently the Grand Central technology based on OpenCL, a newly developed programming language. Its potential scope is parallelization of C-like programming languages for execution on graphics processors. NVidia proposed the CUDA language, very similar to OpenCL, which can be used to translate sequential C code into parallel threads that can be run on NVidia's GPUs and Stream is AMD's similar offering.

The recently announced Prism tool from criticalBlue tackles the same problem of legacy sequential code parallelization. It essentially predicts the application performance under different

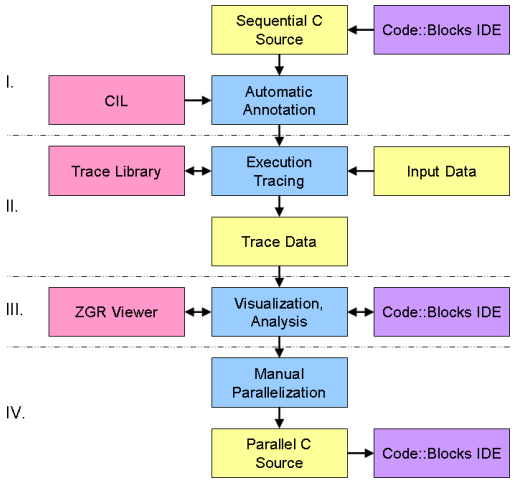

Fig. 1. The toolset flow.

```
*mptr = (*mptr + *qmatrix/2)/(*qmatrix);
```

⇓

```
__cil_tmp9 = *qmatrix;
__cil_tmp10 = __cil_tmp9 / 2;
__cil_tmp11 = *mptr;
__cil_tmp12 = __cil_tmp11 + __cil_tmp10;
*mptr = __cil_tmp12 / __cil_tmp9;
```

Fig. 2. Three-address code rewrite of source complex expressions.

```
iptr = &ivect[i];
heap_write(..., &iptr, 4, ...);
...
ivect[i] = 256;
heap_write(..., &ivect[i], 4, ...);
...
heap_read(..., &iptr, 4, ...);
heap_read(..., &(*iptr), 4, ...);
ivar = *iptr + 12;
heap_write(..., &ivar, 4, ...);
```

Fig. 3. Annotation of instruction data dependencies.

thread decompositions, and the corresponding inter-thread dependencies. Like in our case, its assessment of parallelization opportunities is bound by the quality of the testbench used.

III. Toolset description

The toolset supports software developer efforts to parallelize sequential C programs using the four-stage flow presented in Fig. 1 (source instrumentation, runtime trace collection and compaction, trace data visualization and analysis) based on toolset components: C source annotator, execution tracer, trace data graphical visualizer, and IDE for project development, toolset integration and visualizer-source code cross-reference.

The first stage automatically rewrites the original C source, while adding calls to execution tracer API and preserving the functionality, using a C-to-C compiler based on the CIL infrastructure [11].

After linking with the tracer library, the instrumented program is run in stage two with an input data set (provided by the developer) that should *maximize the discovered dynamic data dependencies* by exercising as many statement-to-statement data dependencies as possible. Execution data are automatically collected and compacted at runtime, and made available at the end of the run to the analysis stage.

Stage three allows the developer to interactively analyze the trace data and execution statistics in a compact graphical format to discover parallelization opportunities, such as pairs of statements or functions with uni-directional data dependencies that can be executed in parallel as stages of a coarse-grained task pipeline. Cross-reference with the sequential C source and other special functions facilitate the exploration.

Manual program parallelization can be done as a new IDE project in stage four, supported by IDE advanced features and the cross-reference with the sequential project and the graphical trace data visualizer.

All toolset components are free software projects. The execution tracer library is a new project in C, the IDE is based on *Code::Blocks* [12] in C++, the C source annotator leverages the *CIL* infrastructure [11] in OCAML, and the trace data visualizer uses the *ZGRViewer* project [13] in Java.

A. Automatic code annotator tool

A special CIL module automatically instruments the source code for program execution tracing. CIL represents a C program using a subset of C syntax and concepts to simplify its manipulation. It is structured as a compiler, with processing modules activated and configured by command-line switches. A Perl wrapper provides a GCC-compatible interface for easy integration in make-based projects.

The code annotation module traverses the CIL representation of sequential program sources and include files, and inserts tracer API calls to collect runtime data of interest. It expects a code with only three-address code instructions (see Fig. 2) and one exit point per function, obtained by activating two modules from the CIL library before the annotation module.

Several source code elements are annotated so that the tracer can keep an accurate program execution trace. This is visualized as a dynamic Data Dependency Graph (DDG) with a node for each program element at an interactively-selectable granularity level (C statement, function, function and its callees). A graph edge links each element that reads data from an address with the element that wrote the last data at that address. For example, the memory location called `iptr` in Fig. 3 determines a dependency edge between the first and third assignment (if no other instruction between the first and second assignment changes the value of `iptr` and no other instruction between the second and third assignment changes the i-th element of array `ivect`) and between the second and third assignment (due to memory location `ivect[i]`, pointed by `iptr`). No dependency is created between the first and second assignment, as they can be executed in any order.

Statement annotations include a unique ID, data address and size, and the source file name and position. Data write annotations include a complexity estimation obtained by adding the weights of statement elementary operations (Fig. 2). Whenever

```
int initVideoIn(THeaderInfo *HeaderInfo)
{
  int t;
  heap_startFunction("initVideoIn", ...);
  heap_arg_write(39, "initVideoIn", 1,
      ..., &HeaderInfo, 4, ...);
  heap_decl(21, ..., &t, 4, 1, 0, ...);
  <function body>
  heap_arg_read(18, "initVideoIn", 0,
      ..., &_retres14, 4, ...);
  heap_endFunction("initVideoIn", ...);
  return _retres14;
}
```

(a)

```
heap_arg_read(54, "getc", 1, ..., &fh1, 4, ...);
ch = getc(fh1);
heap_arg_write(55, "getc", 0, ..., &ch, 4, ...);
```

(b)

Fig. 4. Annotation of a function definition (a) and call (b).

```
static int i = -1;
```

⇓

```
static int i = -1;
...
void initVideoIn(THeaderInfo *HeaderInfo)
{
  ...
  if (!heap_decl_globals_done)
    heap_decl_globals();
  ...
}
...
static void heap_decl_globals(void)
{
  heap_decl(1114, "i", &i, 4, 1, 1, ...);
  ....
  heap_decl_globals_done = 1;
}
```

Fig. 5. Annotation of global variable declarations.

possible, annotations use source variable names instead of temporary variables created during complex expression dismantling.

Function definition annotations (Fig. 4-a) include the entry and the exit points (`heap_startFunction()`, (`heap_endFunction()`)), and data dependencies through the stack (for formal arguments and return value) – the tracer uses a virtual stack controlled by complementary API calls to `heap_arg_write()` (push) and `heap_arg_read()` (pop) in both callee and function call site (Fig. 4-b)[1].

All variable declarations are annotated using calls to `heap_decl()` API (Fig. 4-a, 5) that associate a memory address to a symbolic name: (1) permanently (static variables), (2) during a function call (automatic variables), or (3) between the `malloc()` and `free()` calls (heap variables). Function-scope variable declarations (Fig. 4-a) include a unique ID, variable name, address, size, number of elements (for index analysis through pointer aliases for vector types), storage class, and its source file path and position.

The C syntax does not allow one to annotate global variables directly where they are declared. Hence their declaration API calls are collected in each source file in a function

[1]Functions with a variable number of arguments require a slightly more dynamic handling within the called function body.

```
img->m_buffer = (uint8_t*)malloc(stride*h);
if (NULL==img->m_buffer)
    free(img);
```

⇓

```
img = (RT_Image *)malloc(16U);
heap_alloc(653, img, 16U, ...);
heap_write(653, &img, 4, ...);
...
if (__cil_tmp19 == __cil_tmp17) {
    free(img);
    heap_free(672, img, ...);
```

Fig. 6. Annotation of dynamic memory operations.

(`heap_decl_globals()` in Fig. 5) that is called at the begin of every annotated function in the file until is executed once.

Dynamic memory operations are annotated to track data dependencies through heap memory blocks (Fig. 6). An API call to `heap_alloc()` associates the heap block address with the pointer name and `heap_free()` removes the association – very much like tools such as `purify` or `valgrind` trace the validity of memory accesses.

Control flow expressions (conditional statements and loops) are not annotated as they would uselessly clutter the parallelization guidance based only on data flow dependencies.

Annotated code can be freely mixed with unannotated code (in source or binary form) to accelerate the execution and allow the use of languages unsupported by the annotator, such as C++. However, the analysis scope is limited as runtime data tracing of unannotated parts is not available (e.g., for library functions). This can be a serious problem for functions that take pointer or non-scalar arguments, or use global pointers, e.g., string manipulation functions. For a reliable analysis their source code should be available and annotated.

B. Execution tracer

The execution tracer is implemented as a library providing the API presented in Section III-A. Linked with the annotated program, it collects execution data into a complex data structure during program run.

Since the calling context is essential for the subsequent analysis for parallelization, most collected data are indexed by it (i.e., nodes of the data dependency graph are uniquified by calling context). This is inefficient for heavily recursive code, and requires compressing call chains with multiple occurrences of the same node (called function ID) to the minimum common sub-chain[2].

Each `heap_write()` API call updates the correspnding DDG node data structure with information that includes its computational weight (an execution time estimate) and source code position. As mentioned above, a DDG node is uniquely identified by instruction ID and call stack. The API call also updates the last write DDG node for the corresponding memory address in the tracer data structure.

API calls to `heap_read()` build the list of read dependencies for the next write instruction. Each dependency between

[2]Formally, one can generate the minimum Finite State Machine which recognizes all the call stack strings, and then follow a shortest path on that FSM for each call stack leaf.

```
a = b + c;    /* instruction ID: 10 */
x = y + z;    /* instruction ID: 100 */
r = x + a;    /* instruction ID: 1000 */
```

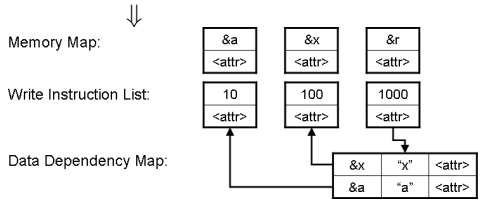

Fig. 7. Creation of data dependencies between instructions.

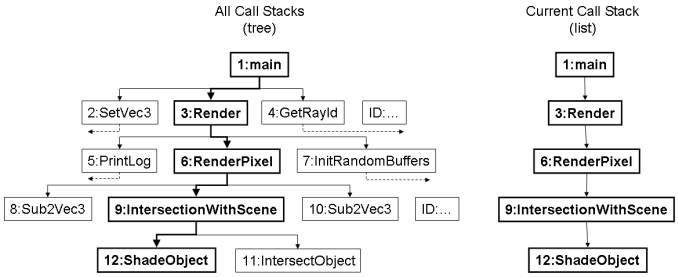

Fig. 8. Data structure to record all call stacks during program execution.

this read and the last previous write is added to the program data dependencies as a DDG graph edge. Thus, each DDG node is associated with a list of DDG nodes that produced its data (Fig. 7).

Function calls and returns are tracked using API calls to `heap_startFunction()` and `heap_endFunction()` that update both the current call stack and the call tree (Fig. 8). The current call stack is a LIFO, while the full call tree is a tree of hash tables where each call level (called function) is associated with a hash table that records the functions it calls.

Data dependencies through the program stack (function arguments and return value) are tracked using the virtual stack explained in Section III-A. Data dependencies between caller actual arguments and callee formal arguments are recorded by the tracer in the same way as data dependencies for instructions.

A symbol table records the base address, size, symbolic name, and source file location for all program variables using an AVL tree [14] indexed by address. Automatic variables are pushed in the symbol table by each `heap_decl()` call and removed at function exit.

At the end of the annotated program execution, the tracer saves data dependencies, call stacks, and other statistics collected in an XML file to be used by the graphic analysis tool.

C. Graphic analysis tool

The huge amount of detail collected during program execution is presented in a very abstract and summarized form, in order to help the developer to focus on data dependencies that lead to parallelization opportunities. The IDE and DDG visualizer are thus very interactive and provide a set of keyboard and mouse actions for efficient data exploration.

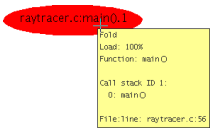

Fig. 9. Most summarized view of program execution trace.

Once the DDG data are loaded, the viewer presents the most summarized representation of program execution (Fig. 9), where the execution of all instructions and all data dependencies are folded into the starting function of the program. The fold name represents the source file, the function name, and its unique call stack ID. Node statistics in the yellow frame indicate node type, that it accounts for 100% of program execution, the call stack up to its function and its source file location.

A "fold" node is a collection (compression) of children nodes such as leaves (elementary C statements) or function calls with or without an entire call tree below them. When folding a node, all data dependencies among its children are hidden, and only dependencies between other leaf nodes or folds and its children are shown. The idea is to *represent what would be the incoming and outgoing data dependencies if this node was chosen as the parallelization unit* (task, thread, Kahn process).

Trace data exploration starts by unfolding this view to display its direct callees (Fig. 10). Data dependencies are represented by directed edges from producer to consumer nodes. Graph element hues go from whitish to intense red and encode the relative execution frequency for nodes (an estimate of amount of computation due to the nodes and their children) and data dependency frequency for edges (an estimate of the amount of data communication between the nodes).

To reduce graph cluttering with many uninteresting folds, the graph can be "re-rooted" to the most significant fold for exploration of parallelization opportunities only within it. Re-rooting assumes that the execution starts on the selected fold and discards everything above it in the call stack. For instance, a re-root to fold `raytracer.c:Render().18` in Fig. 10 would only discard about 0.01% of program execution.

The viewer adds to the standard functions of the underlying ZGRViewer tool (e.g., zoom, pan, magnifier) some DDG-specific functionalities. The latter include a navigatable history of folding and re-rooting states, node bi-directional cross-reference with program source in the IDE, and external data dependency view for functions, which is very important for thread-level parallelization. The latter summarizes the graph to the set of nodes (leaves or folds) that exchange data with a given node, in order to simplify data dependency exploration using viewer-IDE cross-reference. It can be used as the basis to define the set of incoming and outgoing data dependencies for a node that needs to be extracted as a thread.

IV. Toolset use example

This section goes through the four stage flow of the toolset presented in Fig. 1 to find a few parallelization opportunities

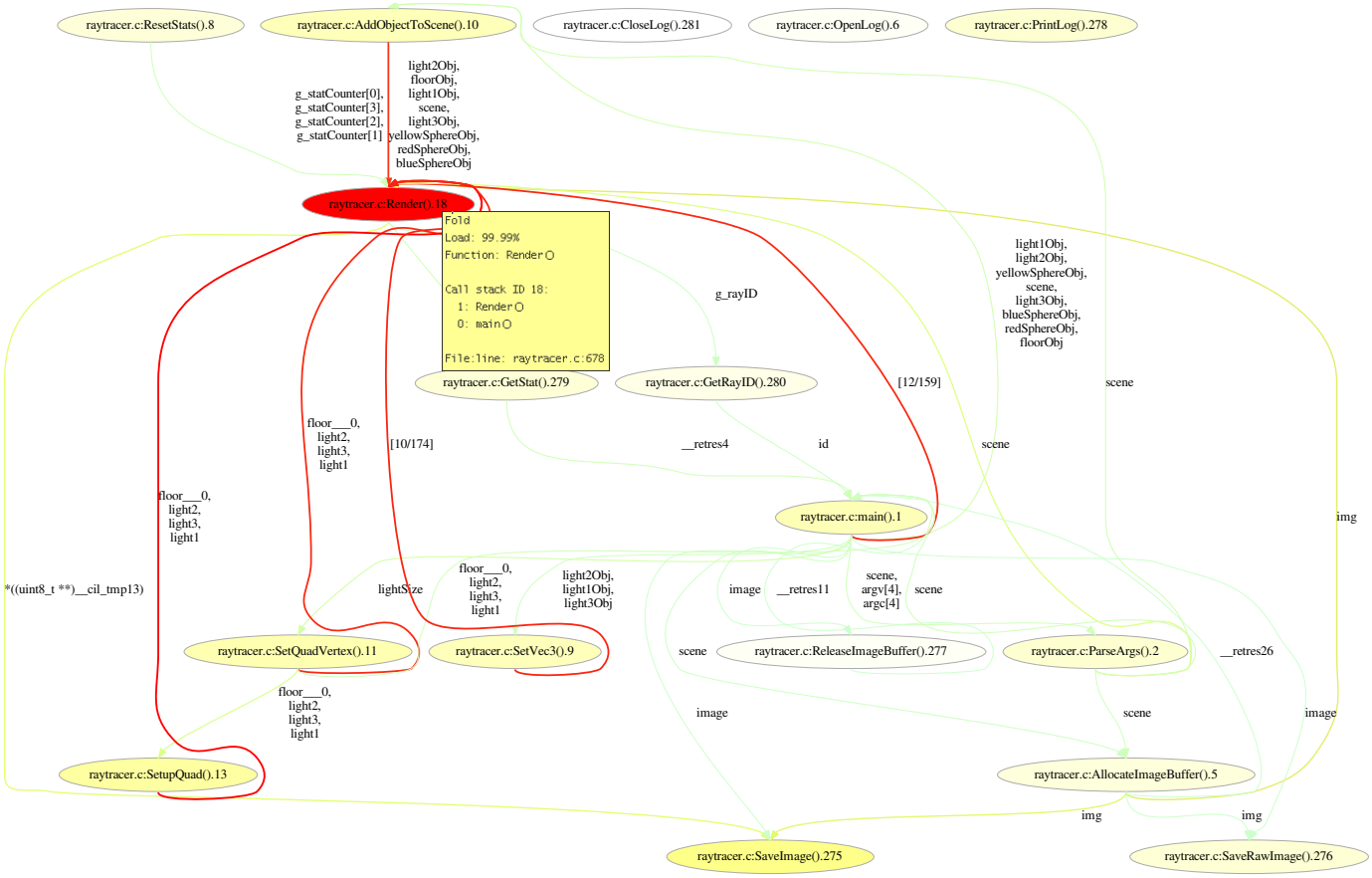

Fig. 10. Trace view expansion to first callees.

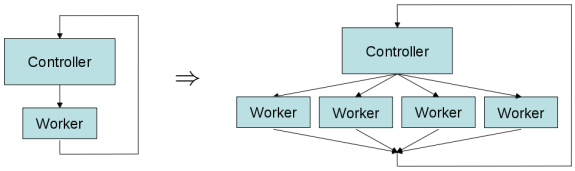

Fig. 11. Controller-worker parallelization.

```
<initilizations>
for (c = 0; c < numClusters; c++)
  for (y = 0; y < blockSize; y++)
    for (x = 0; x < blockSize; x++)
      RenderPixel(scene, x, y, image, c);
<cleanups>
```

Fig. 13. Controller-worker form of `Render()` function.

for a sequential implementation of a ray tracing program.

Ray tracer algorithms model light sources as rays that bounce on scene objects to compute realistic lighting for computer-generated images. Light rays are processed mostly independent of each other, and offer good thread-level parallelization opportunities as multiple parallel workers (Fig. 11).

In the first stage (Fig. 1), the sequential program is imported as a make-based IDE project, with added targets to build the annotated source and instrument the program for tracing.

During the second stage, trace data are collected by running the annotated program with an input set chosen by the developer to exercise program dependencies. This data set must also be small enough to result in an acceptable execution time, considering the 500× slowdown due to annotation.

In the third stage, trace data exploration starts from the top-level view in Fig. 9, unfolded as in Fig. 10. The focus is on folds with most computational load and Fig. 12 a-c show several parallelization opportunities. For instance, function `Render()` and its callees account for most execution in Fig. 10 and calls to function `RenderPixel()` and its callees account for virtually all computational cost of `Render()` unfolded in Fig. 12-a. Data flow analysis shows good data independence for `RenderPixel()`. Moreover, the `Render()` source code (Fig. 13) closely resembles the controller-worker pattern (Fig. 11), with `RenderPixel()` as worker.

However, unfolding further on and cross-referencing with the source code we easily see that a very similar analysis can be applied to function `IntersectionWithScene()` and its workers in Fig. 12-b and 12-c, for which a parallelization example is presented in Fig. 14.

V. Conclusions

The toolset presented in this paper supports manual parallelization of sequential C programs. It automatically rewrites

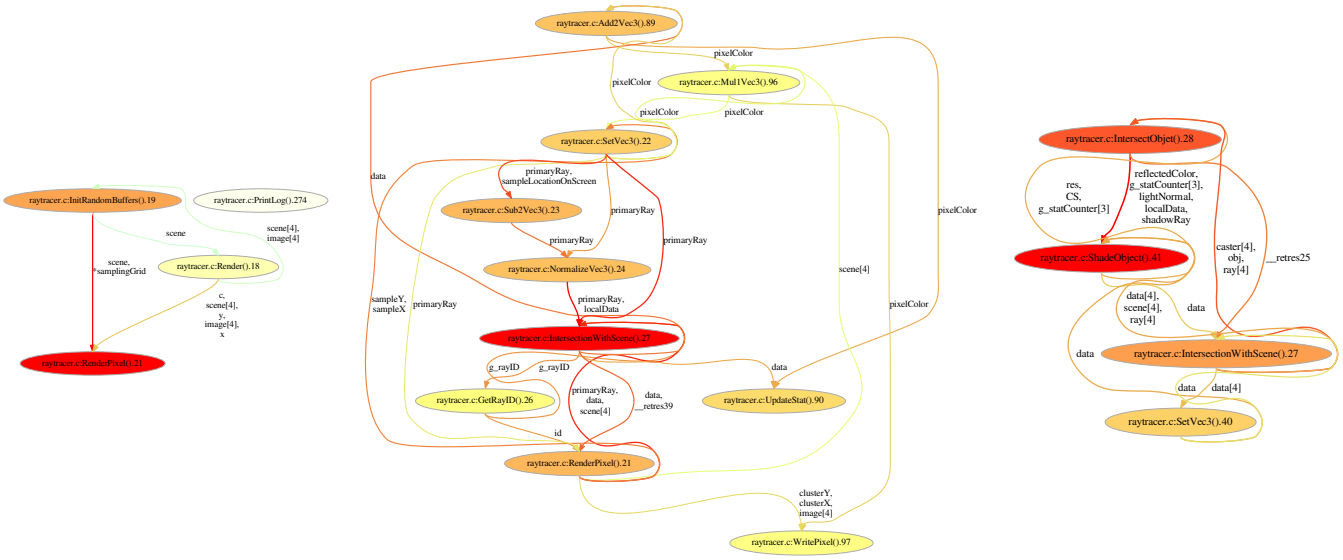

Fig. 12. Sequential unfolding of high-execution folds (b-99.77%, c-99.3%) and applying graph re-rooting.

```
while (obj != NULL) {
  <setup>
  if (IntersectObjet(obj,ray,caster,&localData)==1)
    if (localData.m_distance < data->m_distance)
      *data = localData;
  obj = obj->m_next;
}
```

⇓

```
while (obj != NULL) {
  for (n = 0; n < NWORKERS; n++) {
    <setup for worker n>
    if (obj != NULL) {
      obj = obj->m_next;
      <start IntersectObjet() worker n>;
    }
  }
  for (n = 0; n < NWORKERS; n++) {
    if (<exit code worker n> == 1)
      if (localData[n].m_distance<data->m_distance)
        *data = localData[n];
  }
  if (obj != NULL) obj = obj->m_next;
}
```

Fig. 14. `IntersectionWithScene()` function controller-worker format and parallelization.

the source code with annotations that trace data dependencies and other events during program execution. The developer can analyze the data collected in an interactive graphical form and can use exploration and analysis tools that effectively support the search for any type of parallel code rewriting opportunities, including, but not limited to, the intrinsically race-free Kahn Process Network style.

The tool is part of the EU FP7 HEAP project where its functionality is complemented and enhanced by two other tools: one for automatic parallelization indices using KPNs [1], [2], and one for functional verification of parallelized code.

Acknowledgments

This work is supported by the European Commission in the context of the FP7 HEAP project (#247615). The ray tracing application analyzed in this paper has been kindly provided by ST Microelectronics within the HEAP project.

References

[1] Compaan Design BV, 2012. See http://www.compaandesign.com/.
[2] B. Kienhuis, E. Rijpkema, and E. F. Deprettere, "Compaan: deriving process networks from matlab for embedded signal processing architectures," in *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, pp. 13–17, 2000.
[3] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 356–369, dec. 2007.
[4] J.-Y. Mignolet, R. Baert, T. J. Ashby, P. Avasare, H.-O. Jang, and J. C. Son, "Mpa: Parallelizing an application onto a multicore platform made easy," *IEEE Micro*, vol. 29, no. 3, pp. 31–39, 2009.
[5] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing* (J. L. Rosenfeld, ed.), (Stockholm, Sweden), pp. 471–475, North Holland, Amsterdam, Aug 1974.
[6] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, pp. 345–420, Dec. 1994.
[7] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "Suif: an infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Not.*, vol. 29, pp. 31–37, Dec. 1994.
[8] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Comput. Surv.*, vol. 27, pp. 367–432, Sept. 1995.
[9] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in Split-C," in *Supercomputing '93. Proceedings*, pp. 262–273, nov. 1993.
[10] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D. Cronquist, and M. Sivaraman, "Pico: automatically designing custom computers," *Computer*, vol. 35, pp. 39–47, sep 2002.
[11] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Int'l Conference on Compiler Construction*, pp. 213–228, 2002.
[12] "Code::Blocks IDE." http://www.codeblocks.org/.
[13] E. Pietriga, "A toolkit for addressing HCI issues in visual language environments," *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, vol. 00, pp. 145–152, 2005.
[14] R. Sedgewick, *Algorithms*. Addison-Wesley, 1983. chapter 15: Balanced Trees.