Interactive Trace-Based Analysis Toolset for Manual Parallelization of C Programs

Interactive Trace-Based Analysis Toolset for Manual Parallelization of C Programs

MIHAI T. LAZARESCU and LUCIANO LAVAGNO, Politecnico di Torino

Massive amounts of legacy sequential code need to be parallelized to make better use of modern multiprocessor architectures. Nevertheless, writing parallel programs is still a difficult task. The automated parallelization methods can be effective both at the statement and loop levels and, recently, at the task level, but they are still restricted to specific source code constructs or application domains. We present in this article an innovative toolset that supports developers when performing manual code analysis and parallelization decisions. It automatically collects and represents the program profile and data dependencies in an interactive graphical format that facilitates the analysis and the discovery of manual parallelization opportunities. The toolset can be used for arbitrary sequential C programs and parallelization patterns. Also, its program-scope data dependency tracing at runtime can complement the tools based on static code analysis, and can also benefit from it at the same time. We also tested the effectiveness of the toolset in terms of time to reach parallelization decisions and of their quality. We measured a significant improvement for several real-world representative applications.

Categories and Subject Descriptors: D.1 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Execution Profiling, Data Dependency Analysis, Program Parallelization

Additional Key Words and Phrases: Legacy C program parallelization, source annotation, execution profiling, data dependency analysis, graph analysis, graph abstraction

ACM Reference Format:
Lazarescu, M.T. and Lavagno, L. 2014. Interactive Trace-Based Analysis Toolset for Manual Parallelization of C Programs. *ACM Trans. Embedd. Comput. Syst.* 14, 1, Article 13 (November 2014), 20 pages.
DOI: http://dx.doi.org/10.1145/2638556

1. INTRODUCTION

As processors and systems refocus from the acceleration of the execution of a single-thread to the increase of the overall throughput by means of multiprocessor architectures, there is an urgent need to parallelize massive amounts of legacy sequential code [Burger and Goodman 2004; Athanasaki et al. 2008; Hennessy and Patterson 2012]. 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 challenging task [Mattson et al. 2004; Hwu et al. 2008; Asanovic et al. 2009].

Automated software parallelization has been extensively explored especially at the statement, basic block and loop levels, which are appropriate for VLIW and vector processors [González and González 1998; Ottoni et al. 2005; Goossens and Parello 2013]. By contrast, the tools for exploring the parallelization opportunities at the *task* level, which are best suited for modern multicore processors, are less explored, with some

Author's addresses: M.T. Lazarescu and L. Lavagno, Electronics and Telecommunications Department, Politecnico di Torino, I-10129 Turin, Italy. Email: mihai.lazarescu@polito.it and luciano.lavagno@polito.it.

DOI: http://dx.doi.org/10.1145/2638556

notable exceptions [Kienhuis et al. 2000; Benabderrahmane et al. 2010; Compaan Design BV 2012]. However, most of the latter techniques are so far restricted to specific types of loops and data access patterns.

The toolset described in this article directly addresses these challenges. It helps the software developers to profile and parallelize the existing sequential C-language applications by exploiting top-level parallelism. It synergistically uses and extends past work [Thies et al. 2007; Mignolet et al. 2009], based on *runtime, full-scope* data dependency tracing and sophisticated graph visualization techniques. These allow the code developers to find faster the best *manual parallelization opportunities*. They can use the toolset to quickly detect *possible* parallelization opportunities and to subsequently assess their *effective suitability* for parallelization, complementing the traditional means, e.g., code inspection.

The approach proposed in this article benefits the search for best parallelization opportunities regardless of the style of parallel code writing. These can include, but are not limited to, the intrinsically race-free Kahn Process Network style [Kahn 1974].

Section 2 of the article reviews other approaches and tools for sequential code parallelization. Section 3 presents the toolset flow and the operation of the component tools. Section 4 presents the results of using the toolset for the analysis and parallelization of several real-life applications. Section 5 concludes the article.

2. RELATED WORK

Code parallelization is one of the most widely studied topics in compilers for parallel machines since the 1970s. Most 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 due to the lack of dependencies (`do-all`) or as a software pipeline [Bacon et al. 1994; Wilson et al. 1994; Allan et al. 1995].

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 multimedia applications in the PC world.

For the reasons above, the need for techniques that can assist the developer to manually partition an application beyond the limitations of automated analysis is growing stronger. For instance, of particular interest is the analysis at top program level, as opposed to the innermost loop level [Culler et al. 1993; Kathail et al. 2002].

[Thies et al. 2007] propose a technique similar to the one implemented by our toolset. This approach is extended by our toolset through techniques based on data compression and advanced visualization that provide the developer with effective display and analysis means for the very large amount of data generated by data dependency tracing (e.g., for a large video encoding or decoding application). We consider these techniques of our toolset essential to reduce the time spent by the developer for program analysis while searching for the best parallelization opportunities.

Mixed approaches, based on both static and dynamic data dependency analysis exist [Tournavitis et al. 2009; Vandierendonck et al. 2010]. These are semiautomatic, relying to different extents on developer support to provide hints (e.g., code annotations) for the selection of the optimal parallel solution. However, the developer is offered limited means to manually analyze and improve the solutions proposed by the tools or to manually find better ones. Moreover, these tools directly modify or generate project source code which is seldom acceptable in a large-scale industrial project due to maintainability and debugging concerns.

Several compilation and debugging tools, often based on proprietary extensions of the C language, have also been proposed by leading industrial players. For example, Apple introduced the Grand Central technology based on OpenCL, a newly developed

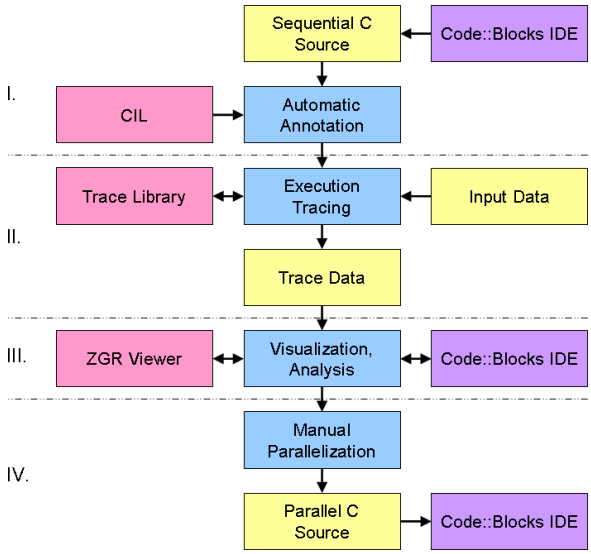

Fig. 1. Toolset flow: I – automatic C source annotation; II – execution tracer; III – data dependency and execution profile interactive visualization; IV – IDE for toolset integration and creation of the parallel project.

programming language. Its potential scope is parallelization of C-like code 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. Stream is a similar offering from AMD. Intel released Parallel Studio to facilitate the development or parallelization efforts by iterating code analysis, execution profiling, race checking and debugging, and performance prediction, but it does not address data dependencies which is a major parallelization inhibitor. Also, more generally, the tools that use static (compile-time) code analysis may miss program-level dependencies that are important to assess parallelization opportunities [Ramalingam 1994; Allen and Kennedy 2002].

Recently announced tools from CriticalBlue and Vector Fabrics tackle the same problem of legacy sequential code parallelization. The former essentially predicts the application performance under different thread decompositions, and the corresponding inter-thread dependencies. The latter uses data-dependency analysis to explore parallelization options for the bottleneck loops in a program. As in our case, the assessment of parallelization opportunities depends on the quality of the profiling data and the tool does not modify directly the project code.

3. TOOLSET DESCRIPTION

The toolset supports the developer efforts to parallelize sequential C programs using the four-stage flow presented in Figure 1 (source instrumentation, runtime trace collection and compaction, trace data visualization, abstraction and analysis) based on the following components: (1) a C source annotator, (2) an execution tracer, (3) a trace data graphical visualizer, and (4) an IDE for project development, toolset integration and display of the visualizer/source code cross-references.

The first stage automatically rewrites the original C source using a C-to-C compiler based on the CIL infrastructure [Necula et al. 2002]. It adds calls to the execution tracer API while preserving the original functionality.

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 discov-*

```
*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. Complex source expressions are rewritten to three-address code (elementary operations).

ered 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 effectively abstract the complexity and size of the trace data and execution statistics. It uses an interactive graphical environment that exploits cross-references to the source code to simplify and accelerate the discovery of parallelization opportunities. These can be at any level and for any parallelization method, such as pairs of code blocks or functions with unidirectional data dependencies that can be executed as stages of a coarse-grained task pipeline.

Manual program parallelization is performed in stage four, supported by the IDE advanced features such as the cross-references between the original source code and the graphical trace data visualizer.

All toolset components are free software projects. The execution tracer library is written in C, the IDE is based on *Code::Blocks*[1] in C++, the C source annotator leverages the *CIL* infrastructure [Necula et al. 2002] in OCaml, and the trace data visualizer uses the *ZGRViewer* project [Pietriga 2005] in Java.

3.1. Automatic Code Annotator Tool

A special CIL module automatically instruments the source code for program execution tracing. CIL builds an internal representation of the C program using a subset of the C syntax and semantics, to simplify its manipulation. It is structured as a compiler, with processing modules activated and configured using 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 the sequential program and inserts tracer API function calls to collect runtime data of interest. It expects a code with only three address code statements (see Figure 2) and one exit point per function. This is obtained by preprocessing the program intermediate representation using two modules from the CIL library, before the annotation module.

Several source code elements are annotated (such as the function entry and exit, function arguments and return value, initialization of the static variables, automatic variable declarations) so that the tracer can keep an accurate program execution trace. This will be visualized in stage three as a dynamic Data Dependency Graph (DDG) with a node for each program element at an interactively selectable granularity level (C statement, C block, function). However, data dependencies are always collected and stored at the statement level. 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 Figure 3 determines a dependency edge between the first and third assignment (assuming that no other statement between the first and second assignment changes the value of `iptr` and no other statement between the second and third assignment changes the i-th element of array `ivect`) and between the

[1]Code::Blocks IDE project http://www.codeblocks.org/

```
1:  iptr = &ivect[i];
    partools_write(..., &iptr, 4, ...);
    ...
2:  ivect[i] = 256;
    partools_write(..., &ivect[i], 4, ...);
    ...
    partools_read(..., &iptr, 4, ...);
    partools_read(..., &(*iptr), 4, ...);
3:  ivar = *iptr + 12;
    partools_write(..., &ivar, 4, ...);
```

Fig. 3. Example of annotations for statement data dependency tracking: block 3 depends on both block 1 and 2, but blocks 1 and 2 are independent. Dependencies on constant values are not traced.

```
int initVideoIn(THeaderInfo * HeaderInfo)
{
    int t;
    partools_startFunction("initVideoIn", ...);
    partools_arg_write(39, "initVideoIn", 1, ..., &HeaderInfo, 4, ...);
    partools_decl(21, ..., &t, 4, 1, 0, ...);
    <function body>
    partools_arg_read(18, "initVideoIn", 0, ..., &_retres14, 4, ...);
    partools_endFunction("initVideoIn", ...);
    return _retres14;
}
```

(a)

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

(b)

Fig. 4. Example of function annotations. The annotations of function definitions (a) include function begin and end, formal arguments, local variable declarations and return value. The annotations of function calls (b) include the function actual arguments and the assignment of the return value.

second and third assignment (due to memory location `ivect[i]`, pointed by `iptr`). The first and second assignment can be executed in any order, thus no data dependency is created between them.

Statement annotations include a unique ID, data address and size, and the source file name and position. Data write annotations include a rough estimation of execution complexity obtained by adding the weights of the elementary operations in the statement (Figure 2). Whenever possible, annotations use the names of source variables instead of the temporary variables automatically created by the C-to-C processor during complex expression dismantling.

For function definitions (Figure 4a), the annotations include the entry and the exit points (`partools_startFunction()`, `partools_endFunction()`), and data dependencies through the stack (for formal arguments and return value)—the tracer uses a virtual stack controlled by complementary calls to the API functions `partools_arg_write()` (push) and `partools_arg_read()` (pop) in both callee and function call site (Figure 4b)[2].

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

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

⇒

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

Fig. 5. The annotations of static variable declarations are collected in a local function. This function is executed only once, before any other function in the source file is executed.

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

⇒

```
img = (RT_Image *)malloc(16U);
partools_alloc(653, img, 16U, ...);
partools_write(653, &img, 4, ...);
...
if (__cil_tmp19 == __cil_tmp17) {
    free(img);
    partools_free(672, img, ...);
```

Fig. 6. Dynamic memory operations are annotated both as heap operations and as writes to the pointer referencing the allocated space.

All variable declarations are annotated using calls to `partools_decl()` (Figure 4a and 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 (Figure 4a) include a unique ID, variable name, address, size, number of elements (for index analysis through pointer aliases for vector types), storage class, and source path and position.

The C syntax does not allow one to annotate global variables where they are declared. Hence their declaration API calls are collected in each source file in a function (`partools_decl_globals()` in Figure 5) that is then called before any function from that file is executed for the first time.

Dynamic memory operations are annotated to track data dependencies through heap memory blocks (Figure 6). A call to `partools_alloc()` associates the heap block address with the pointer name and `partools_free()` removes the association—very much like other tools such as *Purify* or *Valgrind* trace the validity of memory accesses.

Control flow expressions (conditional statements and loops) are currently not annotated, since in our experience they are more local and clutter the parallelization guidance based only on data flow dependencies.

The annotated code can be freely mixed with unannotated code (in source or binary form) to accelerate the execution and allow the use of languages currently unsupported by the annotator, such as C++. However, this also limits the analysis scope, since the 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

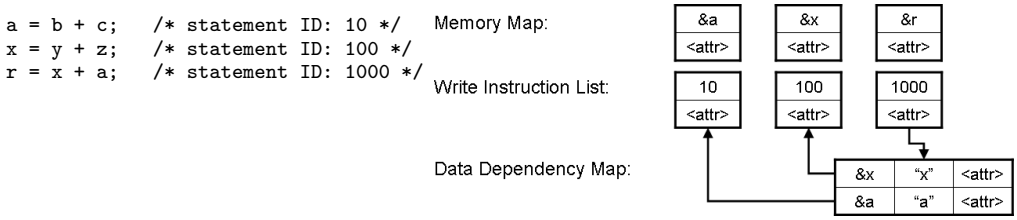

Fig. 7. Example of mapping of the runtime data dependencies between statements on the data structures of the tracing library.

source code or at least a stub illustrating input/output argument data dependencies must be available and annotated.

3.2. Execution Tracer

The execution tracer is implemented as a library providing the API functions presented in Section 3.1. Linked with the annotated program, it collects execution data into a complex data structure during the 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 the 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 subchain[3].

Each `partools_write()` call updates the corresponding 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 the statement ID and the call stack. The call also updates the last write DDG node for the corresponding memory address in the tracer data structure.

The calls to `partools_read()` build the list of read dependencies for the next write statement. Each dependency between this read and the last previous write is added to the program data dependencies as a DDG graph edge. Thus, each DDG node (source code statement) has an edge from each DDG node that produces its input data and an edge to each DDG node that uses its output data (Figure 7).

Function calls and returns are tracked using calls to `partools_startFunction()` and `partools_endFunction()` that update both the current call stack and the call tree (Figure 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 presented in Section 3.1. The data dependencies between the caller actual arguments and the callee formal arguments are recorded by the tracer in the same way as the data dependencies for statements.

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

At the end of the execution of the annotated program, the tracer saves the data dependencies, call stacks, and other statistics in a file in XML notation that is used by the graphic analysis tool.

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

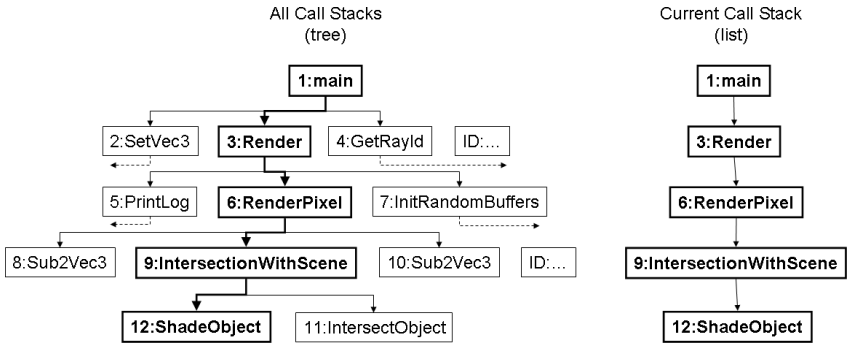

Fig. 8. All call stacks during program execution are recorded in nested hash tables and are labelled with unique IDs. The current call stack is recorded separately as an ordered list.

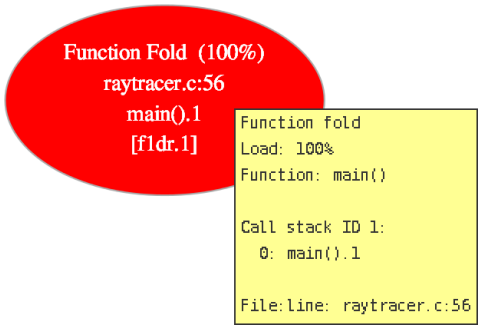

Fig. 9. Initial (most summarized) view of the program execution trace.

3.3. 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 the data dependencies that can 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.

Once the DDG data are loaded, the viewer presents the most summarized representation of the program execution (Figure 9), where the execution of all statements and all their data dependencies are folded into the starting function of the program. The fold name displays the fold type, its execution load, the source file name and line, the function name, and its unique call stack ID. Node statistics in the rectangular frame indicate the node type, the fact 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 a call tree below them. When a node is folded, all data dependencies among its children are hidden, and only the dependencies between other leaf nodes or folds and its children are shown. The purpose is to *represent what would be the incoming and outgoing data dependencies if this node were chosen as the parallelization unit* (task, thread, Kahn process).

The developer starts the interactive trace data exploration by unfolding this view to display its direct callees (Figure 10). Data dependencies are represented as directed edges from producer to consumer nodes.

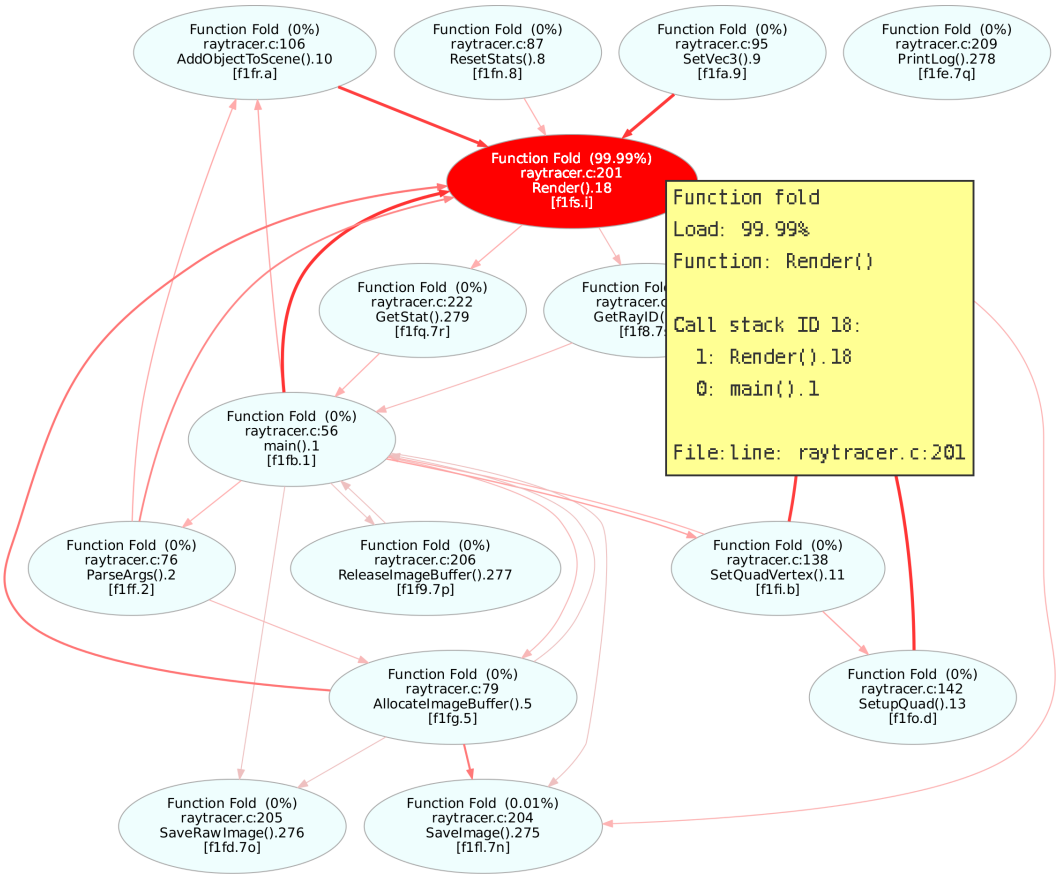

Fig. 10. Trace view expansion to first callees for a ray tracer program.

To facilitate and speed up graph comprehension and exploration, the relative execution frequency for nodes (an estimate of the amount of computation due to the nodes and their children) and the data dependency frequency for edges (an estimate of the amount of data communication between the nodes) is encoded in their visual appearance. The user can switch at any time between color or grayscale encoding. A higher color intensity (or darker shade of gray) and an increased width for edges correspond to higher execution and data transfer frequencies, as can be seen in Figure 10.

Graph comprehension and exploration can be further facilitated and accelerated by applying several filters that can reduce graph cluttering and can help the developer to focus on the analysis on the most important parts for parallelization.

Graph "re-rooting" to a given folding node assumes that the program execution starts on the selected fold, discarding everything above it in the call stack. For instance, a re-root to the fold `raytracer.c:Render().18` in Figure 10 would only discard about 0.01% of the whole program execution time. Each re-rooting also sets the execution load of the new root to 100%, since everything outside it is discarded for the purpose of the subsequent analysis. Re-rooting the graph to this node thus discards many folds with little relevance for parallelization opportunity discovery, but that would end up cluttering the graph. Figure 11 shows how graph re-rooting can prevent excessive graph cluttering during exploration for parallelization opportunities.

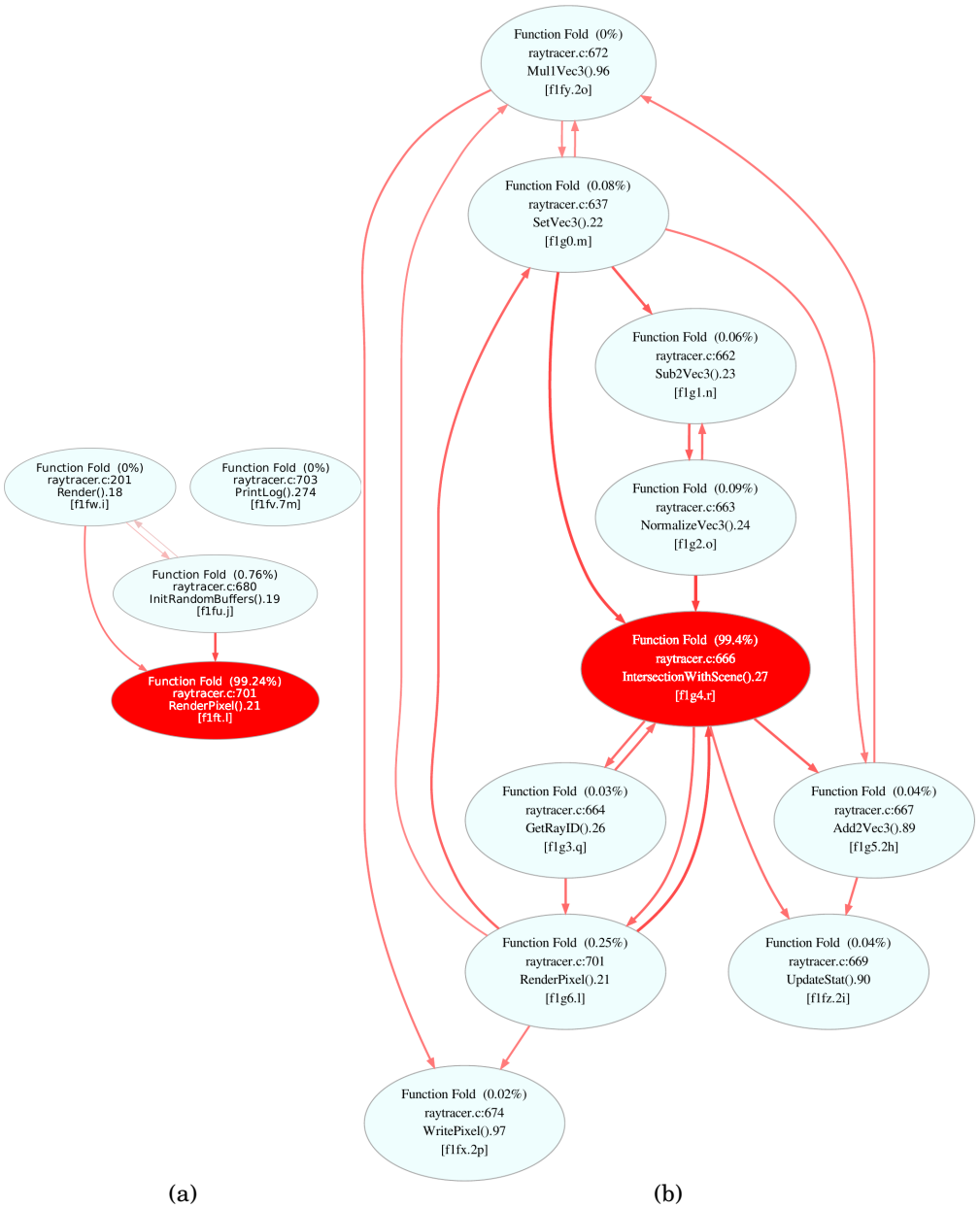

Fig. 11. Sequential unfolding of high-execution folds (a-99.99%, b-99.77%) and applying graph re-rooting.

For the same purpose of helping the developers to focus on the discovery of the most suitable parallelization opportunities, the toolset allows them to filter out of the view the nodes whose estimated execution effort is below a configurable threshold. Moreover, graph cluttering can also be reduced by toggling the edge labels and the amount of information provided by the node labels.

Another important feature for the analysis of the parallelization candidates is the detailed data dependency view of a selected DDG node. The most useful case is when

Fig. 12. Excerpt of the data dependency view of the full `RenderPixel` call stack of the ray tracer application. From top to bottom the layers in the view are: statements *producing* data dependencies *outside* the call stack; the data produced by these; the statements that *consume*/*produce* data dependencies *within* the call stack; the data produced by these; the statements *consuming* the data dependencies *outside* the call stack.

a DDG node is actually a *fold of a whole call stack*, i.e., a collapsed view of a node in the call tree (a given function call instance), its statements, and all the statements in all the functions called by it. For each node in the call tree, the detailed data dependency view represents a summary of the input and output data dependencies of the node itself and of all its callees. It can thus be used to identify the data it receives from and sends to other parts of code outside this call stack. This is an essential piece of information for any parallelization mechanism, language and method and has the structure shown in Figure 12:

— the top layer shows the statement nodes (grouped by functions, shown as rectangular boxes), which *produce* the inbound data dependencies for the call stack under analysis;
— the next layer shows in parallelogram boxes the data produced by these statements;
— the middle layer shows only the statement nodes of the selected call stack that *consume* these data or *produce* data consumed by statements outside the call stack;
— the next layer shows in parallelogram boxes the data produced by these statements;
— the bottom layer shows the statement nodes outside the call stack that *consume* the data produced within the call stack under analysis.

The nodes represent C source statements. Figure 13 shows how their colorization intensity is normalized to the highest execution load in the graph, in order to emphasize the execution hotspots. For instance, the node representing the most executed statement (accounting for 16.62% of the full program execution in the detailed data dependency view shown in Figures 12 and 13) is colorized with full intensity. This eases

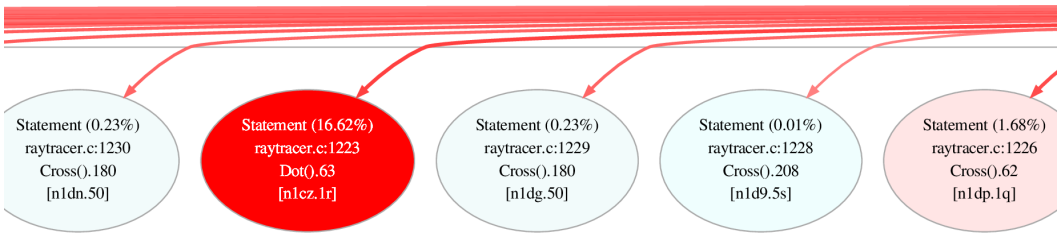

Fig. 13. The colorization of the data dependency view is enhanced to facilitate the visual identification of the execution and data dependency host spots.

spotting nodes with lower execution time by proportionally enhancing all colorizations, as shown in Figure 13.

The detailed dependency data shown in this view are needed for and can substantially speed up the parallelization decisions made by the developers. Moreover, these dependencies are typically difficult to extract by code inspection or analysis using other tools, since the producers and consumers connected by them can be located at various depths in different call stacks, and the dependencies can be over *any storage type* (dynamic, local, global, etc.)

In this regard, it is worth noting that the toolset tracks data dependencies through variables with any scope or storage class, including those that are dynamically allocated on the heap. For instance, Figure 14a summarizes the data dependencies of the call stacks originating within the function `decode_one_frame()` of the H.264 decoder. The toolset allows the developer to further explore them at this level using dependency views like the one shown in Figure 12. However, these dependencies may be difficult to detect using other tools that do not use our fold-based summarization method. This is because the actual statements that generate them can be buried well below this level, in different parts of the code and call stacks, as shown in Figure 14b (in this case only for one dependency variable, `array2D`). This figure shows the program call graph as reported by Callgrind, in which the rectangular boxes denote functions and the arrows connect callers to callees. The thick horizontal line represents the unfold level of Figure 14a and the squares attached to the function blocks show the position in the call graph of the statements that are connected due to the aforementioned `array2D` variable dependencies.

To further help the developers to parallelize the program, the toolset can insert into the source code comments that list the input and output data dependencies and an OpenMP pragma template that can be used for the parallelization of a fold node of interest. Figure 15 shows the comments holding the input and output data dependencies and the OpenMP pragma template generated by the tool, upon user request, for the full `RenderPixel` call stack of the ray tracer application.

To further simplify program exploration, all the nodes in the toolset views, for both statements and data, are cross-linked to the source code in the IDE. This allows an effective graph-driven exploration of the code parts that are considered relevant for parallelization. Also, all views expanded during graph exploration are stored in a history list that facilitates the navigation between several parallelization opportunities.

These DDG-specific functions of the viewer complement the standard graph viewing functions of the underlying ZGRViewer tool (e.g., zoom, pan, magnifier, search).

4. TOOLSET USE

The toolset purpose is to simplify the search for the most promising parallelization opportunities and the selection of the best parallelization method, e.g., data-parallel or

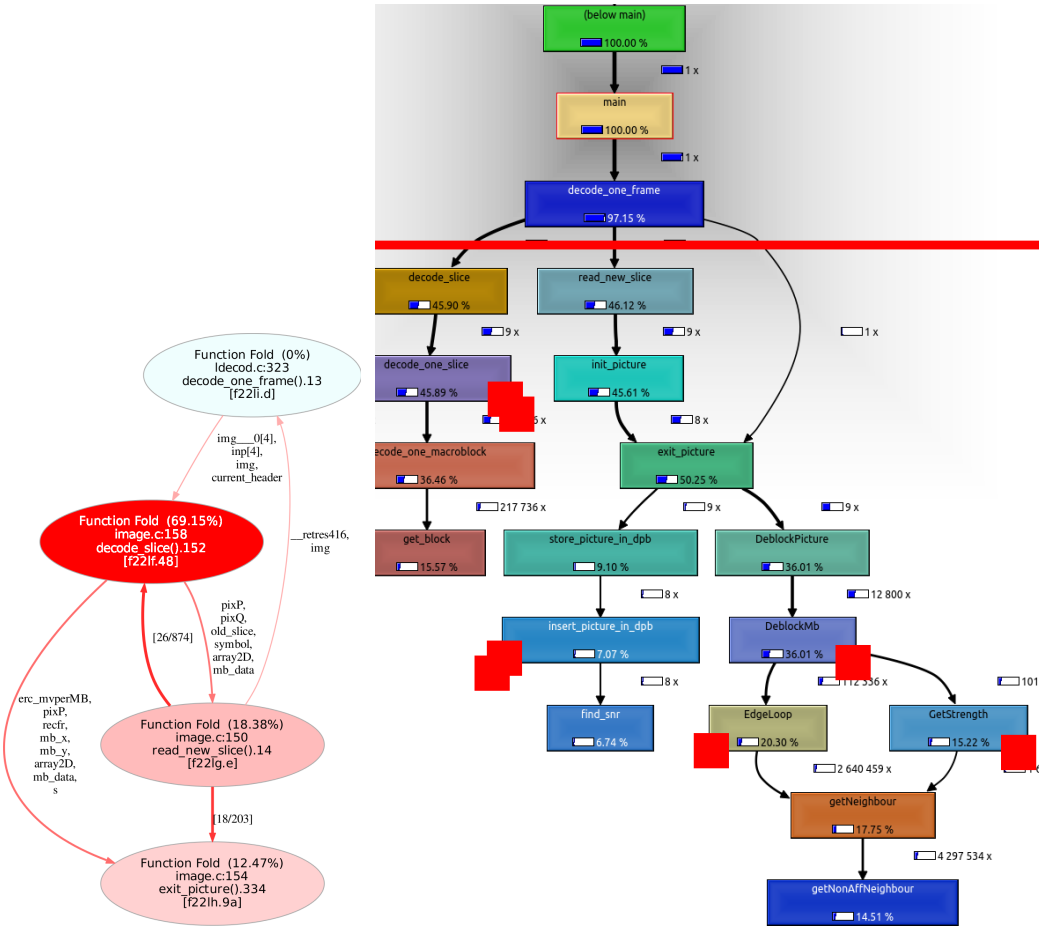

Fig. 14. H.264 decoder data dependencies shown between (a) call stacks are exposed at high levels (e.g., at the level of function `decode_one_frame()`) but are due to (b) statements buried within the call stacks (e.g., the square boxes attached to the function blocks show the position in the call graph of the statements that are connected by data dependencies related to variable `array2D`).

```
            for(y=0; y<blockSize; y++)
                for(x=0; x<blockSize; x++)
// The following statement has:
//
// * input data dependencies:
//   light2Obj, c, img, image, blueSphere, redSphere, blueSphereObj, floorObj, *samplingGrid, light1Obj, ye
//
// * output data dependencies:
//   img->m_buffer
//
// OpenMP pragma template:
// #pragma omp task shared(light2Obj, c, img, image, blueSphere, redSphere, blueSphereObj, floorObj, *sampl
                    RenderPixel(scene, x, y, image, c);
```

Fig. 15. The toolset can insert the input and output data dependencies and an OpenMP pragma template as comments in the source code just above the statement corresponding to a selected node.

```
%     cumulative self    self      total
time  seconds    seconds calls     s/call s/call name
16.61      2.79    2.79 788215425   0.00   0.00 Dot
13.78      5.12    2.32 141631877   0.00   0.00 IntersectQuad
 8.26      6.50    1.39 281277610   0.00   0.00 intersectObject
 8.02      7.86    1.35 139645733   0.00   0.00 IntersectSphere
 7.90      9.19    1.33  69361053   0.00   0.00 NormalizeVec3
 7.69     10.48    1.29 220258108   0.00   0.00 Cross
 6.42     11.56    1.08 268195824   0.00   0.00 Mul1Vec3
 6.12     12.59    1.03 350638670   0.00   0.00 Sub2Vec3
 4.46     13.34    0.75  45257208   0.00   0.00 IntersectionShadowWithScene
 4.22     14.05    0.71 191964084   0.00   0.00 Add2Vec3
 3.77     14.69    0.64 330958926   0.00   0.00 UpdateStat
 3.36     15.25    0.56  15085736   0.00   0.00 CastShadowRay
...
```

Fig. 16. `gprof` output for the ray tracer application.

task-parallel. It does not make any specific assumption on the parallelization method, syntax or tool, even though it was designed originally for task- or process-level (rather than loop-level) parallelism.

Parallelizing an existing sequential implementation *without any prior knowledge* of the software and guided only by a classical source code profiler is not trivial, as argued above. For instance, the `gprof` output for an application may look like Figure 16. It shows clearly the parts of the program where most of the computation occurs, but it does not provide any information on how the *data flow* through the code. More detailed reports from other tools (e.g., from Intel Parallel Studio) still do not provide the means to analyze the data dependencies within the whole program.

However, it is well known that the data flows and dependencies are one of the most important parallelization inhibitors. Thus, the tools should assist the developer to weigh them efficiently and in detail, both locally and at program level.

To illustrate the benefits of the capabilities of the toolset in this respect, we present the results of a comparative use test of the toolset. The purpose of the test is to show how the use of the toolset helps relatively inexperienced personnel to more effectively parallelize a previously unknown legacy application.

We used students from a second-year course for the electronics engineering master (5th year overall). Its purpose is to teach modelling languages, such as SystemC, Esterel and Kahn Process networks, and the associated synthesis and verification algorithms and tools.

The goal of the course is not specifically to teach how to parallelize software. Hence, the students were asked to perform the experiment without any previous knowledge of what writing parallel software means. They had only a generic knowledge that exploiting parallelism is very important in contemporary embedded systems and had used the SystemC language to model multiple threads communicating via signals (i.e., using the Moore synchronous reactive model). They were also exposed to the concept of Kahn Process Networks, but had never written code using this computation model.

The test assignment was to analyze and parallelize three real-life use cases: an MJPEG encoder, a ray tracing algorithm, and a cascade of two FIR filters.

The experiment was carried out using the following main phases:

(1) a 30 minute general presentation of the assignment topic, followed by a request for interest from the students.
    The purpose of this step was to measure their level of interest for software parallelization and how many of them felt confident enough to enrol;

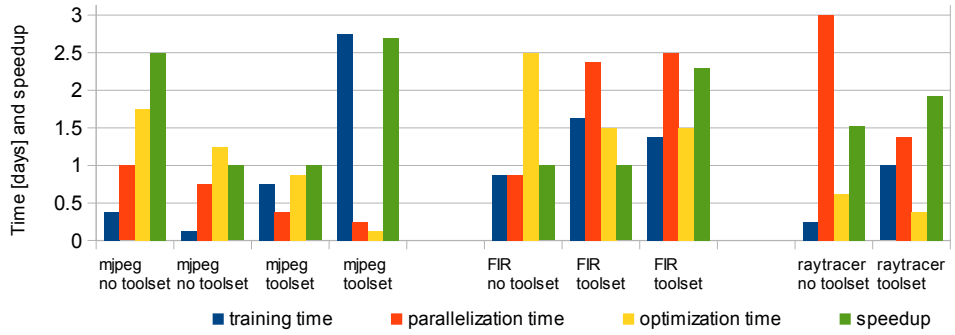

Fig. 17. Results of the test of toolset use for parallelization.

(2) provide the students with some material on software parallelization to become familiar with the main issues and techniques used in the field;
(3) give a full three hour lecture presenting software parallelization. The lecture covered motivations, benefits, limitations, architectural issues, main approaches. The theoretical presentation was followed by an in-class demo of the toolset operation using some simple programs.
At the end of the presentation, the interested students were requested to formally enrol in the assignment. In total 16 students enrolled and were divided into 11 groups of 1–2 students each;
(4) provide one workspace to each group on a virtual machine with four processors. All students were requested to carefully track the time spent working on each phase of the project:
— learning how to use the tools;
— identifying the parallelization opportunities;
— performing the parallelization;
— debugging and optimizing the parallel code.
The groups were partitioned in two sets:
(a) one set was required to perform the parallelization using standard code analysis and development tools, such as `gprof` and a version of `gcc` supporting the OpenMP parallelization pragmas.
We used the results of this set as baseline to assess the effects of using the toolset;
(b) another set was required to perform the parallelization using the toolset described above, in addition to previously mentioned standard tools.
The results of this set were evaluated against the results of the first set separately for each parallelization candidate program, to take into account differences in code structure.
All workspaces included generic software analysis and development tools, the toolset (only for the second set), the code to parallelize, and user documentation.

The students were requested to spend at most a couple of days on the parallelization. Only 9 groups out of 11 completed the assignment.

The results of the test are summarized in Figure 17. The X axis lists the test cases as follows:

(1) “mjpeg” is an MJPEG encoding algorithm with an acyclic data dependency graph at the top level (see Figure 18a);
(2) “FIR” is a couple of cascaded FIR filters (see Figure 18b);

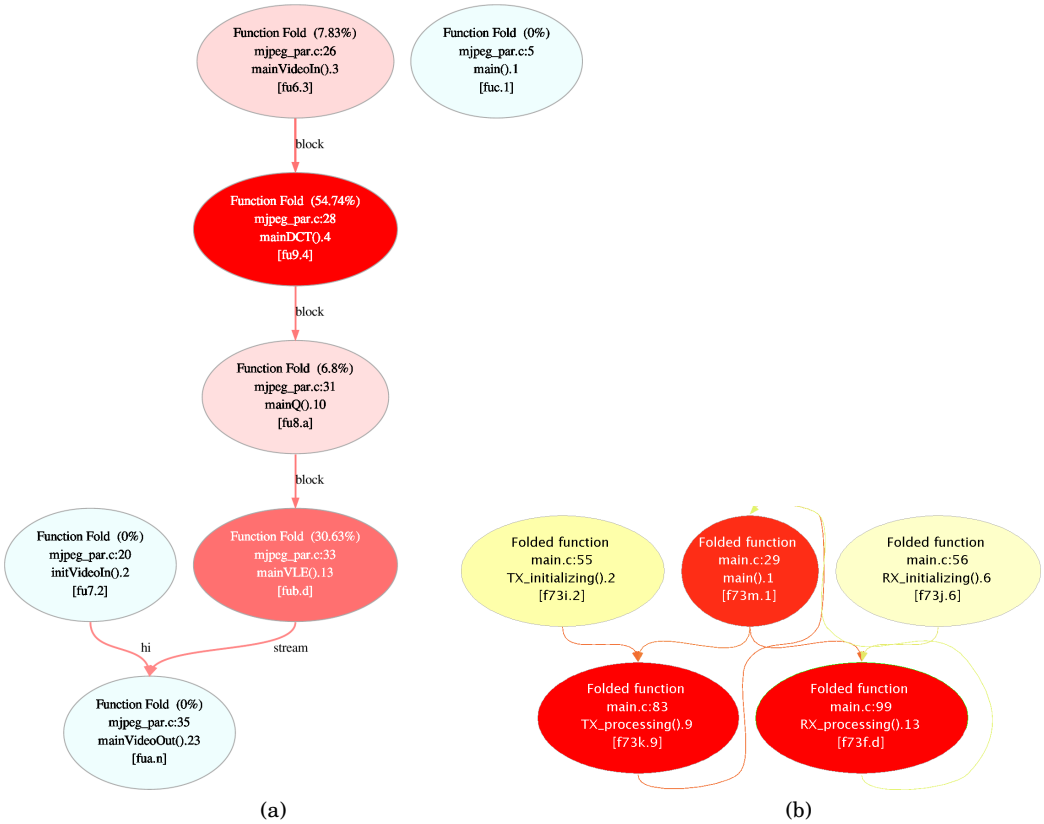

Fig. 18. Top-level execution profile and acyclic data dependency for (a) the MJPEG encoder and (b) the FIR filters (at the bottom, data dependency through `main()` function).

(3) "raytracer" is a ray tracing application with a well known top-level data parallelism (see Figure 10) and architectural effects that limit the parallelization performance, as will be discussed later.

The tools used for the parallelization of each test case are listed as follows:

no toolset. Means that the parallelization was performed using the standard development and analysis tools;
toolset. Means that the parallelization was performed using also the toolset described in this article.

The Y axis shows the time (in days) needed to complete the various phases of the parallelization assignment, and the speedup obtained on a 4 core Intel architecture.

In more detail, the graph indicates:

— the training time to get acquainted with the tools;
— the time to perform the first parallelization (discover parallelism, analyze the data dependencies, write the parallel code using OpenMP pragmas, and debug the results so that the execution was correct);
— the time to further optimize the parallelized code in order to improve the speedup;
— the final speedup with respect to the sequential program.

Based on the results of each test case, we can reach the following conclusions:

(1) "mjpeg" parallelization:
   (a) toolset use considerably reduced the parallelization time, but at the cost of more training time (which is expected given the prototype nature of the toolset);
   (b) the final speedup results with and without tools are similar:
      — one group that used the toolset and one that did not use it managed to obtain some speedup,
      — one group that used the toolset and one that did not use it did not obtain any speedup.
   (c) the parallelization time using the toolset was always shorter than without.
   (d) investing more time to learn the toolset appears to pay off by reducing the parallelization time later.
(2) "FIR" parallelization:
   (a) the group using the toolset was the only one obtaining any speedup;
   (b) learning how to use the toolset in this case took a long time.
(3) "raytracer" parallelization:
   (a) toolset use reduced the parallelization time, at the cost of more training time;
   (b) the parallelization with the toolset had a slightly better speedup than that without the toolset;
   (c) however, neither group obtained a functionally correct parallelization (as discussed below).

The parallelized ray tracer code was not functionally correct for both groups who worked on it, i.e., both groups missed some of the data dependencies due to the incompleteness of their code analysis. This, up to a point, is unavoidable because of the "optimistic", trace-based, manual parallelization approach used. However, after the experiment we extended the toolset with the capability to insert as comment in the source code the list of dependencies for the selected statement and an OpenMP pragma template that can be used to run it in parallel as presented in Section 3.3. Moreover, verification methodologies can be used to identify and debug parallelization errors, as discussed below.

Both groups missed the best parallelization opportunities for the ray tracer and thus achieved less speedup than an experienced programmer would. In fact, an experienced programmer would have added all data dependencies indicated by the toolset to the parallelization solution selected by the student group. So, if the students had used the new version of the tool (with automated OpenMP pragma template insertion) most likely they would have obtained a correct parallelization. The code section selected for parallelization by the group using the toolset is suboptimal, since program execution traverses it many times leading to a high runtime overhead. In fact, after editing the code to use the correct set of dependencies, the speedup with respect to sequential execution reduced to 1.06.

The solution proposed by the group that was not using the toolset included three parallel sections, one of which being the optimal solution. Thus, an experienced programmer would have removed all but the best one, achieving a speedup with respect to sequential execution of 1.67.

Higher speedups, above $2\times$ on four cores, can be obtained by properly handling the well known false sharing effects of the ray tracing algorithm [Yang et al. 2006] and by using a suitable scheduling of the parallel tasks to improve the utilization of the available CPUs. As shown in Figure 19, the load is unevenly distributed over the algorithm iterations leading to significant underutilization of the computing resources unless the tasks are scheduled dynamically or statically with a very fine grain. Neither of these could be achieved by the students within the short time allotted by the experiment.

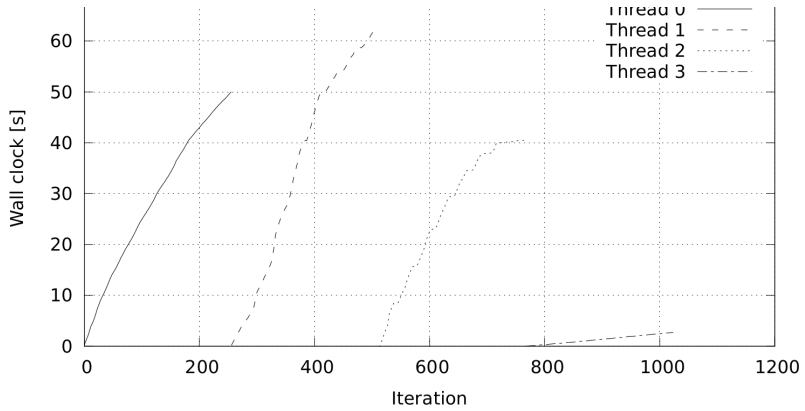

Fig. 19. Time per iteration for ray tracer application parallelized using OpenMP static scheduling.

5. CONCLUSION AND FUTURE WORK

The toolset presented in this article supports the developer analysis and decisions in the very difficult task of sequential C program parallelization.

We illustrated using the results of a test we conducted with MS students how the toolset can help even inexperienced programmers to discover potential parallelism. We found that the interactive graphical profile and *program-wide data dependency* analysis tools effectively support the execution trace exploration and the search for any type of parallel code rewriting opportunities, including Kahn Process Networks, POSIX threads, OpenMP pragmas, and so on.

The application cases also show how the toolset can be used to discover multiple parallelization opportunities, leaving the developer the freedom of choice of which one is best for the target architecture.

It is also worth noting that the toolset can complement existing automatic parallelization tools such as [Compaan Design BV 2012], which can greatly benefit from the toolset-driven *program-wide data dependency* analysis.

In fact, the user of an automatic parallelization tool would need to perform an educated guess to determine which parts of the code to rewrite in order to satisfy the restrictions imposed by the automated tool (e.g., perfectly nested loops and affine array accesses). The toolset presented in this article can provide useful information on:

— where the compute-intensive procedures are;
— if there are any data dependencies besides those through procedure arguments;
— whether the procedure inputs and outputs are truly unaliased;
— whether the procedure inputs are truly read-only and outputs are truly write-only.

Moreover, the use of the toolset on different types of code and by users with various skill levels also exposed some limitations that we intend to address in future work:

(1) merge the profile data from several runs to improve the coverage and accuracy;
(2) import execution profile data from external tools, such as `gprof`;
(3) use a binary-based code instrumentation method to avoid changing the source code, since this can have undesired side-effects like reduced (and different) compiler optimizations and the inability to analyze binary-only code;
(4) more extensive experimentation, using other real life applications;
(5) better support for the designer in the analysis of the trace data, based also on the lessons learned from the student test;

(6) improved support for semi-automated parallelization directive insertion and parallel code rewriting.

These changes have the purpose to extend the current toolset accuracy and applicability to other problems that can also benefit from it.

ACKNOWLEDGMENTS

This work was supported by the European Commission in the context of the FP7 HEAP and PHARAON projects. The ray tracing application presented in this article was kindly provided by ST Microelectronics within the HEAP project, while the FIR application was kindly provided by Thales in the context of the PHARAON project.

REFERENCES

ALLAN, V. H., JONES, R. B., LEE, R. M., AND ALLAN, S. J. 1995. Software pipelining. *ACM Comput. Surv. 27,* 3, 367–432.

ALLEN, R. AND KENNEDY, K. 2002. *Optimizing compilers for modern architectures*. Morgan Kaufmann San Francisco.

ASANOVIC, K., BODIK, R., DEMMEL, J., KEAVENY, T., KEUTZER, K., KUBIATOWICZ, J., MORGAN, N., PATTERSON, D., SEN, K., WAWRZYNEK, J., WESSEL, D., AND YELICK, K. 2009. A View of the Parallel Computing Landscape. *Communications of the ACM 52,* 10, 56–67.

ATHANASAKI, E., ANASTOPOULOS, N., KOURTIS, K., AND KOZIRIS, N. 2008. Exploring the performance limits of simultaneous multithreading for memory intensive applications. *The Journal of Supercomputing 44,* 1, 64–97.

BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26,* 4, 345–420.

BENABDERRAHMANE, M.-W., POUCHET, L.-N., COHEN, A., AND BASTOUL, C. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. In *Compiler Construction*, R. Gupta, Ed. Lecture Notes in Computer Science Series, vol. 6011. Springer Berlin Heidelberg, 283–303.

BURGER, D. AND GOODMAN, J. 2004. Billion-transistor architectures: there and back again. *Computer 37,* 3, 22–28.

COMPAAN DESIGN BV. 2012. See http://www.compaandesign.com/.

CULLER, D., DUSSEAU, A., GOLDSTEIN, S., KRISHNAMURTHY, A., LUMETTA, S., VON EICKEN, T., AND YELICK, K. 1993. Parallel programming in Split-C. In *Supercomputing '93. Proceedings*. 262–273.

GONZÁLEZ, J. AND GONZÁLEZ, A. 1998. The Potential of Data Value Speculation to Boost ILP. In *Proceedings of the 12th International Conference on Supercomputing*. ICS '98. ACM, New York, NY, USA, 21–28.

GOOSSENS, B. AND PARELLO, D. 2013. Limits of Instruction-Level Parallelism Capture. *Procedia Computer Science 18,* 0, 1664–1673. International Conference on Computational Science.

HENNESSY, J. L. AND PATTERSON, D. A. 2012. *Computer architecture: a quantitative approach*. Elsevier.

HWU, W.-M., KEUTZER, K., AND MATTSON, T. 2008. The concurrency challenge. *Design Test of Computers, IEEE 25,* 4, 312–320.

KAHN, G. 1974. The semantics of a simple language for parallel programming. In *Information processing*, J. L. Rosenfeld, Ed. North Holland, Amsterdam, Stockholm, Sweden, 471–475.

KATHAIL, V., ADITYA, S., SCHREIBER, R., RAMAKRISHNA RAU, B., CRONQUIST, D., AND SIVARAMAN, M. 2002. Pico: automatically designing custom computers. *Computer 35,* 9, 39–47.

KIENHUIS, B., RIJPKEMA, E., AND DEPRETTERE, E. F. 2000. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*. 13–17.

MATTSON, T., SANDERS, B., AND MASSINGILL, B. 2004. *Patterns for Parallel Programming*. Software Patterns Series. Pearson Education.

MIGNOLET, J.-Y., BAERT, R., ASHBY, T. J., AVASARE, P., JANG, H.-O., AND SON, J. C. 2009. Mpa: Parallelizing an application onto a multicore platform made easy. *IEEE Micro 29,* 3, 31–39.

NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Int'l Conference on Compiler Construction*. 213–228.

OTTONI, G., RANGAN, R., STOLER, A., AND AUGUST, D. 2005. Automatic thread extraction with decoupled software pipelining. In *Proceedings of 38th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. IEEE.

PIETRIGA, E. 2005. A toolkit for addressing HCI issues in visual language environments. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) 00*, 145–152.

RAMALINGAM, G. 1994. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems 16,* 5, 1467–1471.

THIES, W., CHANDRASEKHAR, V., AND AMARASINGHE, S. 2007. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. 356–369.

TOURNAVITIS, G., WANG, Z., FRANKE, B., AND O'BOYLE, M. F. 2009. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. *SIGPLAN Not. 44,* 6, 177–187.

VANDIERENDONCK, H., RUL, S., AND DE BOSSCHERE, K. 2010. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. ACM, New York, NY, USA, 389–400.

WILSON, R. P., FRENCH, R. S., WILSON, C. S., AMARASINGHE, S. P., ANDERSON, J. M., TJIANG, S. W. K., LIAO, S.-W., TSENG, C.-W., HALL, M. W., LAM, M. S., AND HENNESSY, J. L. 1994. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not. 29,* 12, 31–37.

YANG, C., CHEN, Y., FU, X., LIM, C.-C., AND JU, R. 2006. A comparison of parallelization and performance optimizations for two ray-tracing applications. *Proceedings of HPC&S 6*, 321–330.