HEAP: A Highly Efficient Adaptive Multi-processor Framework

HEAP: a Highly Efficient Adaptive multi-Processor framework

(Invited Paper)

Luciano Lavagno, Mihai Lazarescu
Politecnico Di Torino, Italy
{luciano.lavagno,mihai.lazarescu}@polito.it

Johan Walters, Bart Kienhuis
Compaan Design, Netherlands
{jwalters,kienhuis}@compaandesign.com

Ioannis Papaefstathiou, Andreas Brokalakis
Synelixis Solutions Ltd, Greece
ygp@ece.tuc.gr, brokalakis@synelixis.com

Florian Schaefer
FS Result, Germany
florian.schaefer@fsresult.com

***Abstract*—Writing parallel code is difficult, especially when starting from a sequential reference implementation. Our research efforts, as demonstrated in this paper, face this challenge directly by providing an innovative toolset that helps software developers profile and parallelize an existing sequential implementation, by exploiting top-level pipeline-style parallelism. The innovation of our approach is based on the facts that a) we use both automatic and profiling-driven estimates of the available parallelism, b) we refine those estimates using metric-driven verification techniques, and c) we support dynamic recovery of excessively optimistic parallelization. The proposed toolset has been utilized to find an efficient parallel code organization for a number of real-world representative applications, and a version of the toolset is provided in an open-source manner.**

I. Introduction

Writing parallel programs has traditionally been considered a difficult task, even when parallelism is taken into account from the beginning. Moreover there is an urgent need to parallelize the massive amounts of legacy sequential code so as to increase its performance on processors and systems that refocus from single-thread acceleration to increasing the overall throughput. Automated software parallelization has been tackled extensively at the instruction level and loop level, which are appropriate for VLIW and vector processors. However, only some past work, namely the Compaan approach [1], has actively sought parallelization opportunities at the *task* level, which are most appropriate for modern multi-core processors. A main limitation of the latter techniques is that so far they required loops enclosing the main computational bottlenecks to have very simple internal control (no early exit, limited support for conditionals, etc.), and data access patterns (very limited use of pointers, only affine array indices, etc.)

The HEAP project faces these challenges directly, by developing an innovative toolset that helps software developers profile and parallelize existing sequential implementations by exploiting top-level pipeline-style parallelism. It synergistically uses and extends with respect to past work:

1) The above mentioned Compaan approach [1], [2], extended to support a significantly larger set of control structures (as detailed in Section III), to provide automatic parallelization capabilities based on full *compile-time* dataflow analysis on a *reduced scope* of the application with a *reduced complexity* (in HEAP referred to as the *pessimistic* approach).
2) A novel approach, related to [3] and [4] and described in Section IV, which uses *run-time, full scope* data-dependency tracing and sophisticated graph visualization techniques to enable the code developer to *optimistically* find the best *manual parallelization opportunities*.
3) Coverage analysis and runtime tracing, as described in Section V, to help the developer verify the manually or automatically parallelized code.

The HEAP project also covers the multi-core computer architecture side, by providing innovative cache coherency strategies, described in [5]. They exploit the data access information provided by the parallelization tools mentioned above to provide improved performance at a dramatically reduced cost with respect to current directory-based methods.

The key observation that allows such an improvement is that explicit classification (e.g., by means of a specific encoding of some bits of the address) of each load or store as operating on private or shared data leads to effective use of a write-through policy for shared data, and of a write-back policy for private data. Since all communication in the parallel model of computation considered in this paper, namely Kahn Process Networks (KPNs [6]), is via single-reader single-writer FIFOs, this classification can be readily performed. In particular, all local process variables are private, while all FIFO variables (indices and buffers) are shared. Since most of the data accesses are on private data, the more efficient write-back policy can be applied for a majority of the accesses, thus providing the claimed cache performance and cost gains.

Note that the HEAP approach is *not limited to a KPN-style parallelism*. Only the automated parallelization path uses it, while manual parallelization (helped by the tools described in the rest of this paper) can use any style of parallel code writing. In this paper, we will consider KPNs because they are intuitive and, thanks to their deterministic behavior

independent of the execution timing and schedule, they make manual parallelization easier and less error-prone.

II. Related Work

Code parallelization is one of the most widely studied topics in compilers for parallel machines since the 1970's. Most of the work so far has focused on the identification of code sections within innermost loops (Fortran "do" or "for" and "while" in C) which can be executed fully in parallel ("do-all") due to the lack of dependencies, or on a vector machine, or as a software pipeline [7], [8], [9]. However, the level of parallelism that can be identified using these techniques is very limited, since it can be significant only for special applications (physics, fluid dynamics, structure engineering), and cannot fully exploit the current architectures dominantly used for gaming and multi-media applications.

On the other hand, there is a strong need for techniques which can help the developer to manually partition an application, going beyond the limitations of automated analysis. For example, we would like to work at the "main" program level (as opposed to the innermost loop level) [10], [11].

One of the most effective automated approaches so far, which is part of the background research of this proposal and will provide us with the "lower bound" to the amount of parallelism, is the Compaan project run at the University of Leiden [1] which is being commercialized by Compaan Design. The Compaan approach focuses on Static Affine Nested Loop Programs (SANLP), which use affine loop bounds and index expressions on non-aliased data. Originally the technology was based on Matlab syntax as input specification and currently it is based on ISO C.

A similar approach, but oriented to the discovery of parallelism in the outermost loops, is used by the PICO high-level synthesis software, currently commercialized by Synopsys [11].

Another similar technique, proposed recently by Amarasinghe and others, provides the basic idea of the "upper bound" to the discovery of parallelism [12]. In HEAP, we have extended it by providing further techniques, based on data compression and advanced visualization, to show the very large amount of data that can be provided by a full data trace of, e.g., a large video encoding or decoding technique.

Several compilation and debugging tools, often based on dedicated proprietary extensions of the C language, have also been proposed by dominant players in the industry. For example, Apple has recently introduced the Grand Central technology, and based it on a newly developed programming language called OpenCL. The potential scope of this approach is to parallelize C-like programming languages for execution on graphics processors.

Besides investing millions of research dollars into the search for a magic fully automated parallelizing compiler or reviving an older language, chip vendors are coming up with stopgaps. Unfortunately, these stopgaps are focused solely on their own silicon. Nvidia has released CUDA in order to help translate C languages into parallel instructions that can be used by Nvidia's GPUs for scientific computing (very similar to Apple's OpenCL), while AMD also has its own similar offering called Stream.

The recently announced Prism tool from criticalBlue, which focuses on the same problem of parallelization of legacy sequential code, is bound to the quality of the provided testbench, in order to assess the parallelization opportunities. Moreover, this tool mainly displays the performance of an application when used under different thread assignments together with the corresponding dependencies in each case. HEAP certainly goes far beyond that since it will provide both lower bounds (using array index analysis) and metric-driven coverage analysis to enhance the "quality" (i.e., coverage) of the testbench itself, as well as interaction with the cache-coherency protocols utilized, etc.

III. Static array-based data dependency analysis

This section provides a short overview of how the Compaan compiler [1] performs its data dependency analysis and identifies sections of code which can be safely executed as concurrent Kahn Process Network (KPN [6]). In KPNs, processes are allowed to communicate only via single-reader single-writer FIFO queues with blocking read semantics, thus ensuring deterministic behavior by construction, regardless of the execution timing.

Compaan's exact dataflow analysis operates at the procedure level (reduced scope) and performs its analysis on C code that adheres to the reduced complexity of Static Affine Nested Loop Programs (SANLP). The Compaan compiler converts a SANLP into an efficiently pipelined KPN. The more repetitive the original program, the more effective the Compaan approach is. Especially applications in the domain of video, telecom and imaging can be easily fit in the SANLP format. This provides a productive approach to convert these applications into multithreaded, streaming implementations.

Within SANLP, control flow decisions and data access patterns depend on compile-time known values, i.e., static affine expressions. Therefore, a SANLP comprises the following:

- C loops equivalent to the form `for(it = e1; it < e2; it += e3) {...}` where `it` is the integer iterator, `e1`, `e2` are static affine (loop invariant) expressions and `e3` is a constant.
- if-then-else statements in the form `if([!] e1 [<, <=, ==, >=, >, |, &] e2) {...} else {...}` where `e1`, `e2` are static affine expressions, i.e., all boolean expressions based on static affine values.
- Statements that read, write and compute locally declared unaliased scalars `s` and/or unaliased multi-dimensional arrays `a[ ]`, `b[ ][ ]`, `c[ ][ ][ ]`... indexed with expressions that are static affine. A statement can consist of procedure calls and standard unary or binary C operators.

Expressions of the form: $c_0 * it_0 + c_1 * it_1 \ldots + c_i * it_i + c$ are static affine if c_0, c_1, ..., c_i, c are constants and it_0, it_1, ..., it_i are one of the following:

- Static affine nested loop iterators

- Run-time constants
- One of the following pseudo-linear expressions, where s is static affine and c is constant:
  - $s\%c$
  - s/e
  - $div\_floor(s, c)$
  - $div\_ceil(s, c)$
  - $max(s, s)$
  - $min(s, s)$

All code *outside* the procedures analyzed by the Compaan compiler does not need to be SANLP. Data dependencies between procedures *called* by the SANLP need to be explicit through their function arguments, rather than sharing any global data.

For example, let us consider the following SANL program:

```
int a[10], b[10][10];
for (int i = 0; i < 10; i++) {
  a[i] = Function1();
  for (int j = i; j < 10; j++) {
    b[i][j] = Function2(a[i]);
  }
}
```

This example shows first the definition of the array variables `a[]` and `b[][]`. This is followed by two nested loops with two enclosed function calls. The for-loops define the loop iterators `i` and `j`. The function call `Function1` is placed before the inner for-loop, which results in a non-perfect nested loop. Compaan can handle such non-perfect nested loops. In the example, all exchanges of data between the function calls are through the arrays `a[]` and `b[][]`. The indexing of the arrays is expressed in linear combinations of the loop iterators `i` and `j`. Actual computations are hidden by the functions `Function1()` and `Function2()`.

Given a SANLP, Compaan can analyze the data dependencies between any pair of statements using the theory described in [2], and also show them graphically using its GUI, as shown in Fig. 1.

For example, given the following code:

```
void accumulator2d(
  short data_in[MAX_I][MAX_J],
  int data_out[MAX_J])
{
  int i, j;
  short a[MAX_I][MAX_J];
  int sum[MAX_J];          // Partial sum

  // Initialize the sum array
  for (j = 0; j < MAX_J; j = j + 1) {
    sum[j] = 0;
  }

  // Stream in data_in and accumulate
  for (i = 0; i < MAX_I; i = i + 1) {
    for (j = 0; j < MAX_J; j = j + 1) {
      a[i][j] = data_in[i][j];
      accumulate(a[i][j],sum[j],&sum[j]);
    }
  }
```

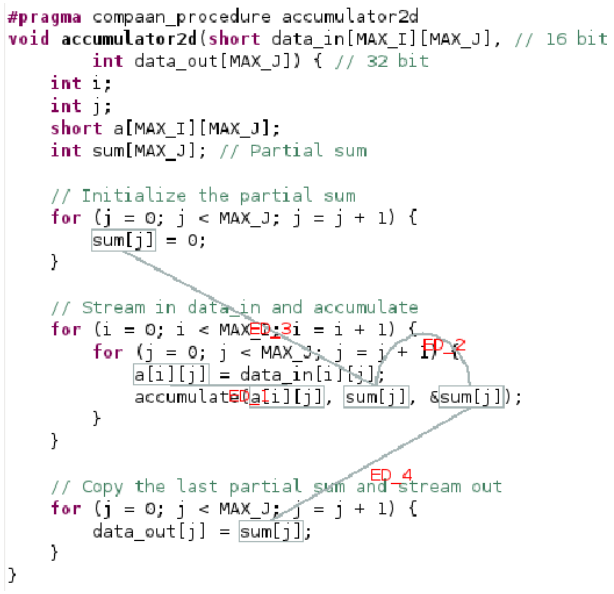

Fig. 1. Data dependencies overlayed on source code by Compaan GUI

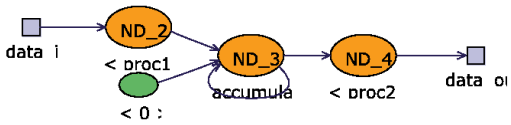

Fig. 2. Example of KPN for the `accumulator2d` function

```
  // Copy the partial sums and stream out
  for (j = 0; j < MAX_J; j = j + 1) {
    data_out[j] = sum[j];
  }
}
```

Compaan derives for this single-threaded, global memory code the KPN described in Fig. 2. Each vertex in the graph represents a statement in the source code and will be implemented as a separate thread. The data dependencies (expressed as edges) are converted into FIFO communication channels. The Compaan compiler automatically produces a KPN for each SANLP and implements the KPN on a multithreaded environment based on pthreads or Intel Thread Building Blocks (TBB).

IV. Dynamic trace-based data dependency analysis

Static analysis techniques, as argued in the previous section, can help the developer automatically parallelize data-intensive code, with limited support for control structures or memory access modes beyond affine indices within uniform vectors. While many embedded applications fall into this category, there is a large amount of legacy software which includes a significant amount of control and decisions, or uses pointers and dynamic memory allocation intensively.

The optimistic software parallelization toolset has been developed specifically to address this second class of ap-

plications. It can be applied to any existing sequential C-language code and helps the software developers to profile and parallelize it by exploiting top-level pipeline-style parallelism.

The parallelized code considered in this paper (only for the sake of easier illustration, as mentioned above) uses the KPN model of computation as the code produced automatically by the tools described in Section III. This model ensures deterministic behavior with arbitrary parallel process execution times, i.e., completely avoiding data races, in order to ease the verification task discussed in Section V. Note that even though in general the deadlock-free executability of a KPN model in finite memory is undecidable, a KPN derived from the parallelization of an existing reference sequential implementation is guaranteed to be schedulable.

The sequential code is manually split (as illustrated in Section VI) into multiple sequential processes that are assigned to parallel resources of the architecture and use FIFO channels for inter-process communication.

Basically, the tool operation consists in the acquisition at run time of several execution data, such as the execution frequencies and data dependencies between the program instructions, as depicted in Fig. 3. This is achieved by annotating the C

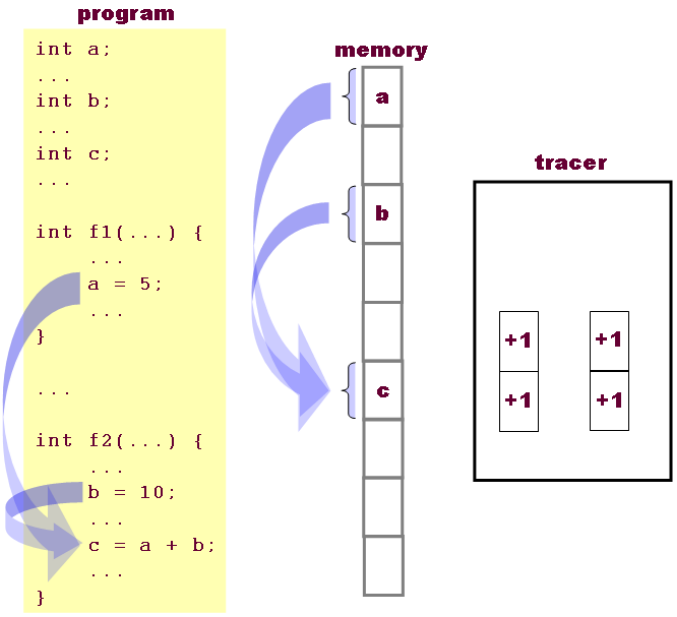

Fig. 3. Basic operation of the data dependency tracing tool.

source with data dependency profiling API calls, as follows:

- `heap_enter_function(char *funcName, int sourceLine, ...)` and `heap_exit_function(char *name, int sourceLine, ...)` used to trace the call stack.
- `heap_declare(char *varName, int sourceLine, void *address, ...)` and `heap_alloc(int sourceLine, void* address, ...)` used to trace the address of static, automatic and dynamically allocated variables. For the first two categories, the name is the same as in the source code. For the latter category, the name is dynamically generated upon every execution of the memory allocation call (based on the source code line where it occurs).
- `heap_read(int sourceLine, void *address, ...)` and `heap_write(int sourceLine, void *address, ...)` used to trace at runtime the reads and writes to an address performed by a statement.

Note that the tracing technique completely solves the aliasing issue. For example, assume that the following source code:

```
1    int a, *b;
2    a = 2;
3    b = &a;
4    ... = *b;
```

is annotated as follows[1]:

```
int a, *b;
heap_declare("a", 1, &a);
heap_declare("b", 1, &b);

a = 2;
heap_write(2, &a);

// heap_read(3, &&a);
b = &a;
heap_write(3, &b);

heap_read(4, b);
... = *b;
```

The dependency is correctly identified as going from line 2 to line 4 of the original code, through variable `a`. Line 3 does not generate any read dependency, since `&a` is effectively a constant at that point of the code, and `&b` is not read any further in the code fragment.

The processing of the API calls at run-time results in the collection of data dependencies, where each dependency is a pair *(producer_statement, consumer_statement)* and is annotated, to help the designer reason about the code structure, with the name (and index in case of arrays) of the source variable through which the dependency occurs. This process of dynamic data flow graph creation is illustrated in Fig. 3.

At the end of the execution, the aggregated data are displayed in an interactive graph cross-referenced with the original source code and which is used to discover and analyse the parallelism opportunities (see Fig. 4). In this graph every node corresponds to a statement in the original source, and every arc corresponds to a set of addresses (labeled with the declared variable name, if applicable) written by the source node statement and read by the sink node statement.

The resulting graph can obviously be very complex, and

[1]For the sake of simplicity we consider a very simple source line identification mechanism here. In reality the HEAP annotator uses both line and column to precisely identify source code locations.

the HEAP Graphical User Interface provides sophisticated mechanisms to:

1) collapse graph nodes at the block and function level (i.e., all the nodes belonging to a block or function become a single node, with all dependencies correspondingly accumulated). Fig. 5 shows an example of function-level collapsing.
2) accumulate dependencies into caller nodes, like the `gprof` tool does for execution times. In this mode, data dependencies between statements of called functions (properly uniquified based on the call tree) are attributed to the callers when the developer requests so.
3) focus on a function (as will be shown in Section VI) and walk over the statements that read data produced by other graph nodes and write data consumed by other graph nodes.

V. Parallelization verification

The methodology used for verification of the optimistic parallelization described above is based on annotations added to both the initial sequential ("golden") version of the program and to the automatically or manually parallelized version. Note that also in case of automated parallelization, which is correct by construction, verification is desirable in order to discover and correct tool bugs.

The annotations produce at runtime a log file that contains data about the program execution that is then analyzed by the analysis tool. The annotation statements are provided by a verification API library. The places where a parallelization tool (or a human developer) changes the code to split the program into multiple parallel sections are the same places where the annotation API calls need to be added in order to track the program state before and after a parallel section. Consequently, the information required to parallelize a program is sufficient to also add the annotations.

Coverage analysis is performed by checking whether every checkpoint in the program has been encountered. If a checkpoint is encountered, the surrounding code has been executed. By placing a checkpoint in every branch of the code, this allows to verify that all branches have been executed. In order to perform that check, the analysis tool requires the checkpoint API calls in the program code, a structure file giving the analyzer a list of all checkpoints and possibly multiple resulting log files to check. The latter might be required because depending on the program structure it might not be possible to visit every branch of a program with a single run – especially if error conditions are to be checked. In this case, a successful standard run plus several runs to cover corner cases might be required to achieve full coverage. In these cases, multiple log files can be provided to the analysis tool and coverage will be calculated over all of them.

The following steps need to be performed in order to verify the parallelized version and the sequential version of a program:

- **segmenting the program** – dividing the program into logical areas (each purely parallel or sequential) by adding the corresponding API calls,
- **dumping data** – adding API calls to the program to dump the input and output data for later comparison,
- **annotating the program** – adding API calls to identify checkpoints, assertions and so on,
- **compiling and running** – this will produce the required coverage and dump data,
- **analyzing the results** – running the analysis tool with the previously obtained log files as inputs.

An example of the annotations on the initial *sequential version* of a hypothetical program (closely resembling the basic structure of the ray tracing application described in the next Section) is as follows:

```
int input[SIZE];
int output = 0;
// initialize input
...
// dump input and start parallel tracing
heap_report_data("area_1", "in", input);
heap_report_start_parallel("area_1");
// Iterate to perform some computation
for (i = 0; i < 10; i++) {
    char task_name[32];
    sprintf(task_name, "task_%i", i);

    heap_report_start_task(task_name);
    ... = input[i];
    // Do some work
    ...
    result = ...
    // Gather the result.
    output += result;
    heap_report_end_task(task_name);
}
// end of area
heap_report_end_parallel("area_1");
// dump output
heap_report_data("area_1", "out", output);
...
```

An example of the annotations on the *parallel version* of the same code is as follows:

```
int input[SIZE];
int output = 0;
// initialize input
...
// dump input and start parallel tracing
heap_report_data("area_1", "in", input);
heap_report_start_parallel("area_1");
// scatter and gather the data
for (i = 0; i < SIZE; i++) {
    FIFOin[i].put(input[i]);
}
for (i = 0; i < 10; i++) {
    output += FIFOout[i].get();
}
// end of area
heap_report_end_parallel("area_1");
// dump output
heap_report_data("area_1", "out", output);
...
// function executed by the i-th process
void process_func(int i)
```

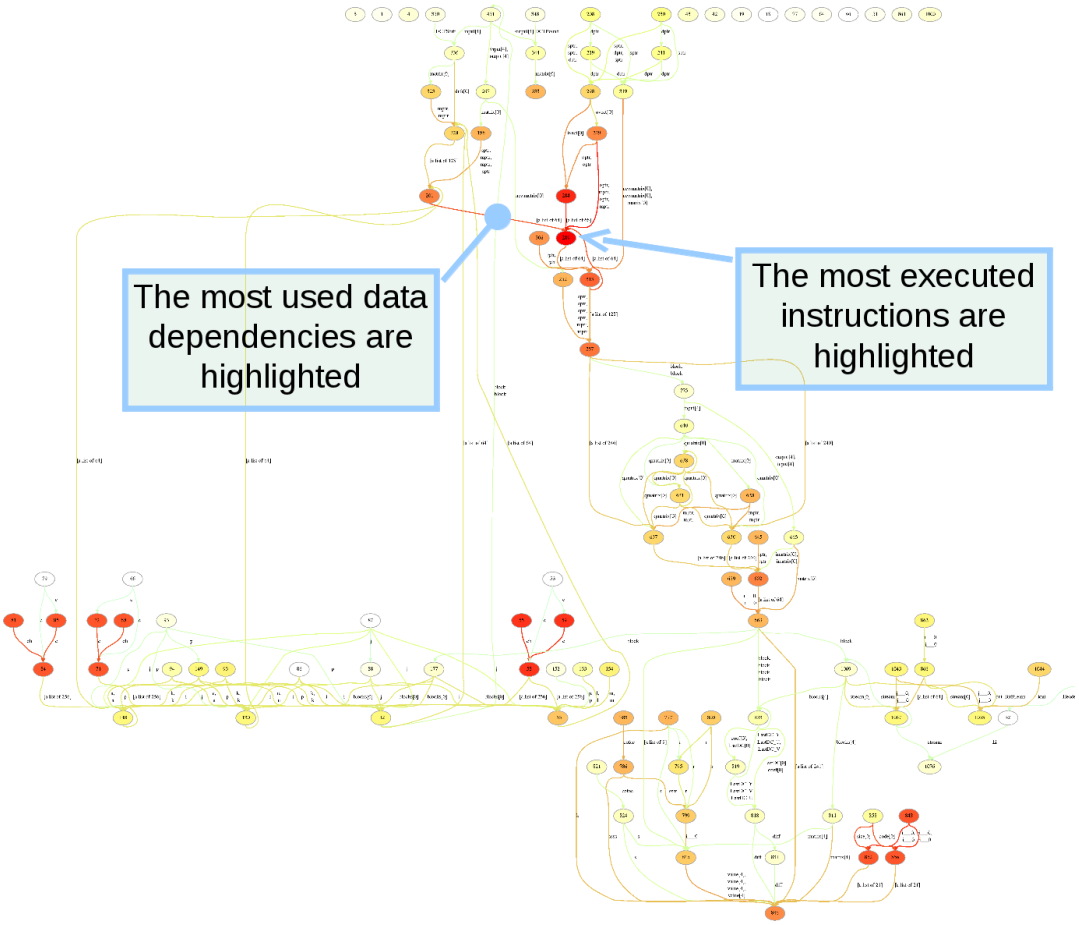

Fig. 4. The graph displays the program instructions as nodes and their data dependencies as directed edges.

```
{
    sprintf(task_name, "task_%i", i);
    heap_report_start_task(task_name);
    ... = FIFOin[i].get();
    // do work
    ...
    result = ...
    FIFOout[i].put(result);
    heap_report_end_task(task_name);
}
```

This example illustrates all previously mentioned steps. The execution of both versions of the code generates both unordered checkpoints (starting and ending of tasks) and data tracing points, which help identifying possible incorrect parallelization results, due to human or tool errors.

VI. An example: parallelizing a ray tracing application

Ray tracing, which mimics the visual process by simulating light rays from light sources, to objects, to the eye, is a widely known "embarrassing parallel" application. However, taking an existing sequential implementation *without any prior knowledge* of the software, guided only by a classical source code profiler can be a daunting task. The `gprof` output may look like this:

```
%      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  MullVec3
 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
...
```

however, it does not provide any clue about how the *data flows* through the code. On the other hand, the output of the HEAP data dependency profiler, shown in Figure 5, shows a clear uni-directional data flow through some of the functions that are at the top in the cumulative profile above, namely `intersectObject`, `intersectQuad` and `intersectSphere`. This provides the developer with the required clue about where to focus the parallelization efforts, on loops involving these functions. These uni-directional data dependencies are essential to highlight both pipeline and do-all parallelism, because they clearly identify *stateless parallelizable computation*.

A quick code inspection shows that `intersectObject` is called exactly in two contexts, with essentially the following code structure:

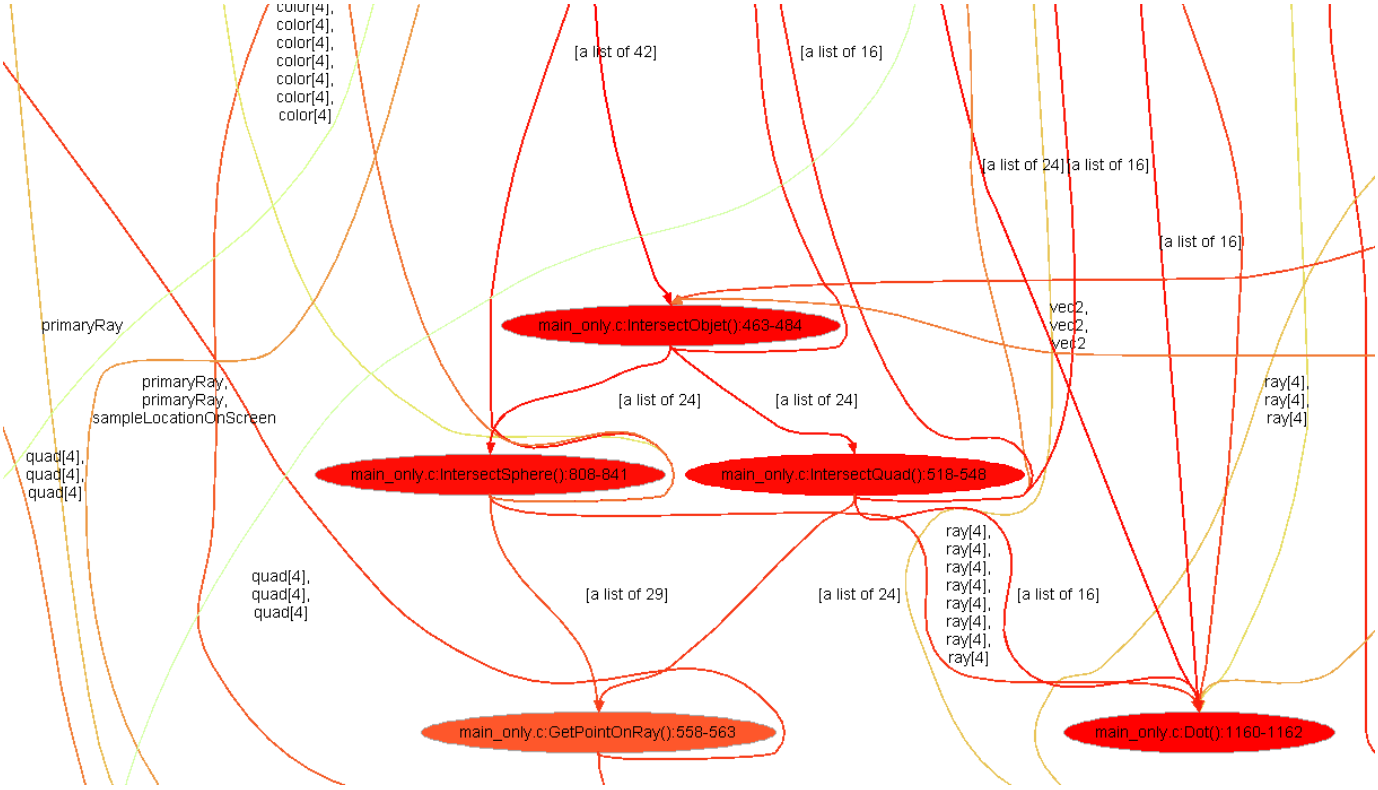

Fig. 5. Data dependency traces around the `intersectObject` function of the ray tracer.

```
  RT_Object *obj = (RT_Object *)scene->m_firstObject;

  while (obj != NULL) {
    localData.m_distance = RT_MAX_FLOAT;
    localData.m_hitFlag = 0;
    localData.m_hitObject = NULL;

    if(intersectObject(obj,ray,caster,&localData)==1)
      if (localData.m_distance < data->m_distance)
        *data = localData;
    obj = obj->m_next;
  }
...
int intersectObject(const RT_Object *obj,
                    const RT_Ray *ray,
                    const RT_Object *caster,
                    RT_IntersectionData *data)
{
  ...
```

In this case, the choice of parallelization *even without any prior knowledge of the application* is quite obvious. One can create a pool of worker tasks, each implementing exactly the same functionality, namely a call to `intersectObject` with `obj`, `ray`, `caster` as inputs and `localData` as output.

Note that even though the inputs to `intersectObject` are `const` pointers, this is no guarantee a priori that they are only used as inputs, since both C and C++ notoriously allow one to cast away `const`-ness and subsequenty *update* the data structures. Regardless, the HEAP data profiler allows the precise identification (within the limitation of the execution paths driven by the provided input data, of course) of which pointers are accessed as inputs and outputs. In this case, it shows (through a more detailed inspection of the profiling data, available through the HEAP graphical user interface) that the inputs are indeed only read and the output only written.

Assuming a FIFO-based KPN structure for parallelization and assuming a goal of N-way parallelization, to match the parallelism of an N-way core, the code above can be changed to the following form:

```
  FIFO(RT_Object) objIn[N];
  FIFO(RT_Ray) rayIn[N];
  FIFO(RT_Caster) casterIn[N];
  FIFO(RT_InterSectionData) dataOut[N];
  FIFO(int) resultOut[N];
  RT_Object* obj = (RT_Object*)scene->m_firstObject;

  while (obj != NULL) {
    // Scatter outputs
    for (i = j = 0; obj != NULL && i < N; i++, j++) {
      objIn[i].put(*obj);
      rayIn[i].put(*ray);
      casterIn[i].put(*caster);

      obj = obj->m_next;
    }
    // Gather inputs
    for (i = 0; i < j; i++) {
      localData = dataOut[i].get();
      if (resultOut[i].get() == 1)
        if (localData.m_distance < data->m_distance)
          *data = localData;
    }
  }
...
void intersectObjectProcess(int i)
```

```
{
  while (1) {
    RT_Object *obj = objIn[i].get();
    RT_Ray *ray = rayIn[i].get();
    RT_Object *caster = casterIn[i].get();
    int result;
    RT_InterSectionData localData;

    localData.m_distance = RT_MAX_FLOAT;
    localData.m_hitFlag = 0;
    localData.m_hitObject = NULL;
    result =
      intersectObject(obj, ray, caster, &localData);

    resultOut[i].put(result);
    dataOut[i].put(localData);
  }
}
```

In the above code snippet, we assume that the runtime system creates `N` concurrent processes, each executing the code of the function `intersectObjectProcess`, each with a different value of `i` from 0 to `N-1`.

This parallelization:

1) can be obtained very quickly. *The entire process, including the debugging, took less than two hours for a programmer with no previous knowledge of the ray tracing application*.
2) is guaranteed to be correct *as long as the only communication occurs via the FIFO queues*.

The latter can be observed by analyzing the data dependency information, but of course can never be guaranteed, because it may be violated along some execution paths which were not traversed due to a limitation of the input data provided to the profiler.

Compaan parallelized the ray tracing application by choosing a different procedure, after some code rewriting in order to improve its automated parallelism discovery. The Compaan parallelization is for a loop performed over all shadow rays, while the manual parallelization is for a loop performed over objects. Both are reasonable candidates, and the best choice depends on the relative number of iterations, which can be readily discovered by source code instrumentation.

However, it is interesting to observe that the Compaan compiler can greatly benefit from the HEAP profiler. The user of the Compaan compiler will need to do an educated guess on which part to rewrite. Typically, these are compute intensive parts which already resemble SANLP, but the HEAP profiler may provide useful information on:

- where the compute intensive procedures are
- whether there are no data dependencies other than through procedure arguments
- whether procedure inputs and outputs are truly unaliased
- whether procedure inputs are truly read-only and outputs are write-only

This ray tracing application case study shows how the HEAP approach can be used to discover multiple parallelization opportunities, leaving to the developer the choice of the one which best suits the underlying multi-core architecture.

VII. Conclusions and future work

This paper described a flexible multi-paradigm approach to the very difficult task of software parallelization. We discussed how potential parallelism can be identified starting both from a formal automated analysis of array indices within loops and from a data dependency execution profile. We explained how both the code changes required to apply the first approach and the manual parallelization changes required by the second approach can be verified by using a metric-driven approach. Finally, we illustrated with a simple but realistic example, a ray tracing application, how different parallelization options can be obtained and quickly explored with the HEAP approach.

Future work will include (1) more extensive experimentation, using other applications provided by the partners of the HEAP project, (2) analysis of the actual speedup obtained by parallelization, including the effects of the HEAP cache architecture, and (3) better support for the designer when rewriting the code for manual parallelization or easier analysis by automated tools, such as Compaan.

Acknowledgment

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

References

[1] C. D. 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. P. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in *40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 356–369, 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] S. Kaxiras and G. Keramidas, "Sarc coherence: Scaling directory cache coherence in performance and power," *IEEE Micro*, vol. 30, no. 5, pp. 54–65, 2010.
[6] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of IFIP Congress*, Aug. 1974.
[7] 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.
[8] 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.
[9] 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.
[10] 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.
[11] 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.
[12] 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.