

Finding Dependencies using Embla

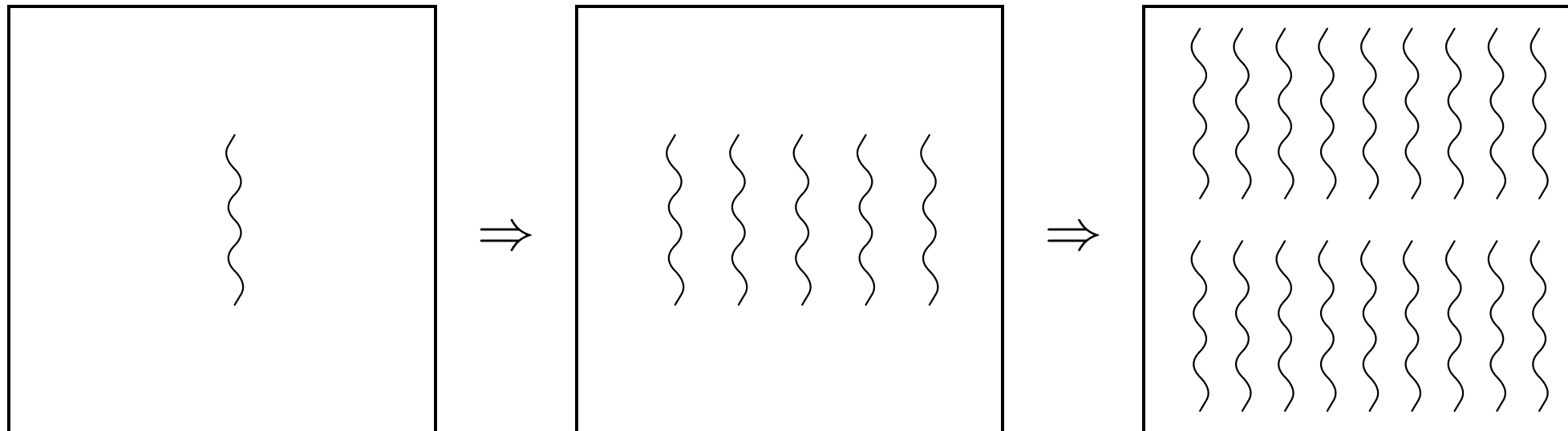
Karl-Filip Faxén

Swedish Institute of Computer Science

Multicore Days 2008

The problem

From single threaded program to scalable multithreading.

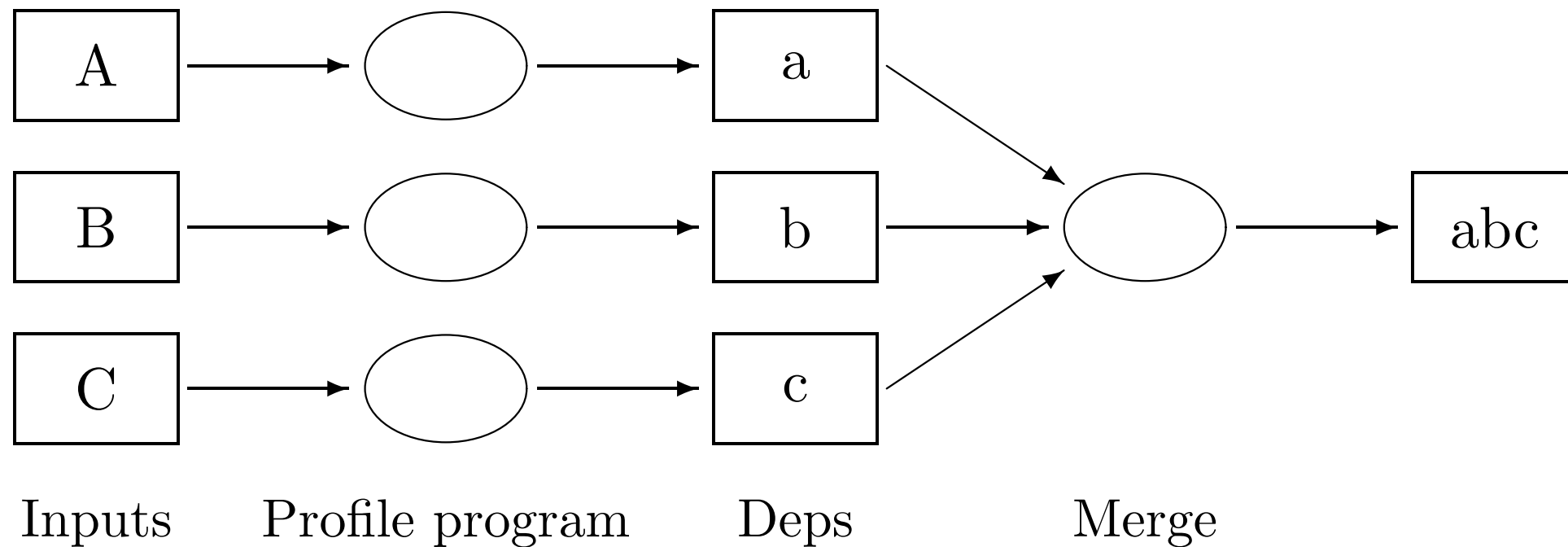


Dynamic Off-line Data Dependence Analysis using Embla

- Support for programmers doing hand parallelisation
- Identifies program parts likely to be *independent* and thus possible to execute in parallel
- Records dependencies *dynamically* in a running program
- Results valid for *same* input, like testing
- Prototype implementation called Embla using the Valgrind instrumentation infrastructure—works for all x86 binaries under Linux

Embla workflow

Run the program several times under Embla with different inputs and merge the profiling output.



```

1:      #include <stdlib.h>
2:      #include <stdio.h>
3:
4:      static void inc(int *p)
5:      {
6:          *p=*p+1;
7:      }
8:
9:      int main(int argc, char **argv)
10:     {
11:         int *q=NULL,n=0;
12:
13:         q = (int*) malloc( sizeof(int) );
14:         inc(q);
15:         inc(&n);
16:         inc(q);
17:         printf( "%d\n", *q+n );
18:         q = (int*) malloc( sizeof(int) );
19:         return q==NULL;
20:     }

```

The diagram illustrates the control flow of the provided C code. It consists of nodes (dots) and directed edges (arrows). A vertical red arrow on the left side of the code block indicates the main execution path from line 1 to line 20. A blue arrow points from the node at line 11 to the node at line 13, representing the call to malloc. A green arrow points from the node at line 16 back to the node at line 14, representing the loop structure of the inc function call. Red arrows also show the flow from line 13 to 14, 14 to 15, 15 to 16, 16 to 17, 17 to 18, 18 to 19, and 19 to 20.

Using the results

Transform program by adding

- explicit `pthread`s,
- lightweight threads (eg filaments),
- OpenMP pragmas
- Cilk-5 annotations (`spawn` and `sync`)
- ...

Why *dynamic* analysis?

- Source code not always available for entire program (legacy, libraries).
- More than one source language might be used.
- Better precision (static analysis always conservative, finds all dependencies and then some).
- As a complement to static analysis—approximate dependencies from below as well as from above.

Why not just insert threads by hand?

Well known debugging problems with hand written threads:

- Explosion of control state space,
- nondeterminism (irreproducibility of bugs).

Difficult to get correct!

Why not just insert threads by hand? (cont)

If parallelism is introduced *systematically* based on dynamic dependence information from Embla:

- Incorrect parallel execution only if a dependence was missed.
- Dependencies are a feature of the sequential program.
- Missed dependencies can be found by debugging in a deterministic, sequential environment (rerun Embla with new input).

Do we find all dependencies?

- Different inputs may yield different dependencies.
- We ran SPEC CPU 2006 benchmark 403.gcc under Embla with 84 different inputs.
- We compared dependencies discovered with different subsets of the inputs.
- We also measured execution coverage using gcov.

Relating execution coverage and dependences found

- We randomly select a set of inputs S and an input j such that
 - $j \notin S$, and
 - every line in gcc executed for j is executed for some $i \in S$.
- Let $D(j)$ and $D(S)$ be the dependencies found for input j or for any input in S , respectively.
- Average size of $D(j) \setminus D(S)$ is 28.5 (average number of dependencies for one input is about 57K).
- 4% of $D(j) \setminus D(S)$ sets were empty (no dependencies missed by looking at S rather than j).

Can we do better?

Transitive filtering

- If we have dependencies $l_1 \rightarrow l_2$ and $l_2 \rightarrow l_3$, then $l_1 \rightarrow l_3$ is redundant, so we can remove it.
- With elimination of transitively redundant dependencies we can do again the above analysis.
- Average size of $D(j) \setminus D(S)$ is 6.2 rather than 28.5 (average number of dependencies for one input is now about 37K).
- 6% of $D(j) \setminus D(S)$ sets are now empty.


Can we do even better?

Using control flow

- The data dependencies computed by Embla do not take control flow into account.
- For instance, the parser of gcc contains a large switch in a loop, and we find many dependencies between different alternatives in a switch. These tend to vary between executions.
- We did control flow analysis by hand for four modules in gcc (among them the parser and the instruction recognizer, which is similar).


Combing with control flow information

```
      f( cond1 )
      a();
      else
      b();
      if( cond2 )
      c();
      else {
      d();
      e();
      }
```



The source and target of the dependence do not have the same control dependence, so they are remapped ...

Combing with control flow information




```
f( cond1 )
  a();
else
  b();
if( cond2 )
  c();
else {
  d();
  e();
}
```

...to the nearest enclosing constructs with the same control dependence.


Combing with control flow information

```
      f( cond1 )
      a();
      else
      b();
      if( cond2 )
      c();
      else {
      d();
      e();
      }
```



Here, only one end of the dependence will be remapped:

Combing with control flow information



```
f( cond1 )
  a();
else
  b();
if( cond2 )
  c();
else {
  d();
  e();
}
```

Combing with control flow information

```
f( cond1 )
  a();
else
  b();
if( cond2 )
  c();
else {
  ↓   d();
      e();
}
```

Here, no remapping is made.

Effect of control flow remapping

- We look only at dependences in the four selected modules.
- With control flow remapping and elimination of transitively redundant dependencies we get the following:
- Average size of $D(j) \setminus D(S)$ is 0.1 rather than 28.5 (average number of dependencies for one input is now about 2K).
- 92% of $D(j) \setminus D(S)$ sets are now empty.

We're doing quite well!

Future work

- Automate control flow analysis
- Analysis of threaded code
- Using Embla to parallelize real programs
- Measuring available parallelism

Measure parallelism

Estimate parallelism implied by the dependencies found

- Shows most profitable sites to parallelize

