TFP – Time-Sensitive Flow-Specific Profiling at Runtime

Sagnik Nandy
Xiaofeng Gao
Jeanne Ferrante

University of California, San Diego

Focus

- Runtime profile-driven optimization
- For runtime use we might make use of the notion of **Persistence**
  - ***K-Persistence*** : A property of the flow that holds true for **K** consecutive execution of the flow

```
for (i=1 to 100)
{
        if(i%2 = 0)
                f();
        else
                g();
}
```

(a)

```
for (i=1 to 100)
{
        if(i > 50)
                f();
        else
                g();
}
```

(b)

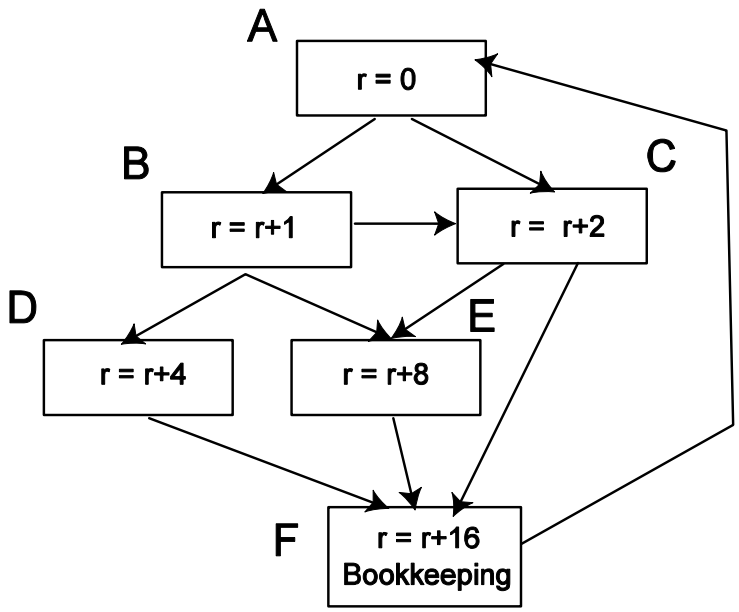
- Gather range of persistent information efficiently

Outline

- **The TFP framework for detecting K-Persistence**
  - **Overview of approach**
  - **Specific Cases**
  - **Examples**
- **Experimental Results**
- **Conclusions, Afterthoughts and Future Work**

TFP – Overview

- Profiling technique for loop bodies
- Each basic block represented as an unique power of 2 (a fixed bit in a bit stream)
- Each basic block ORs this value to a global identifier
- Bookkeeper at exit block digests information

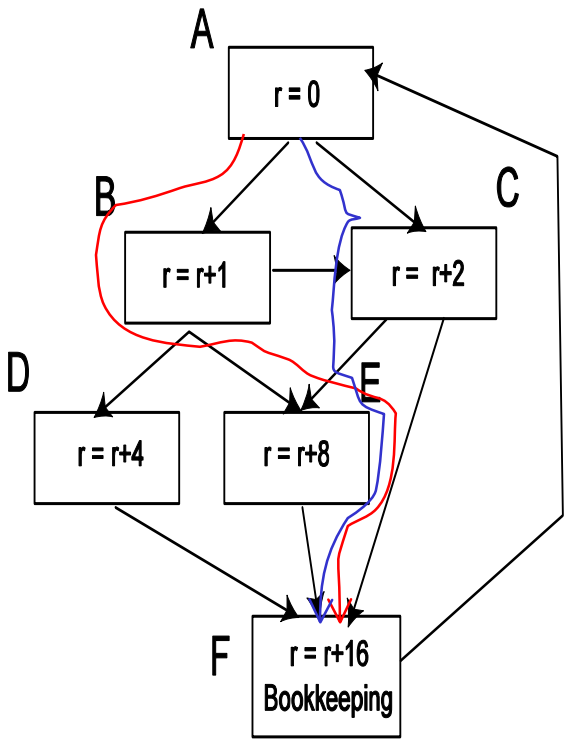


ACF	- 10010
ABEF	- 11001
ABDF	- 10101
ABCF	- 10011
ACEF	- 11010
ABCEF	- 11011

TFP – Specific Cases

- **Persistent Paths**
  - Bookkeeper maintains 2 variables – r_{OR} and r_{AND} which OR and AND the values of ***r*** to themselves
- **Persistent Path Segments**
  - Same bookkeeper as above; block values should be assigned in topological sorted order
- **Persistently taken/un-taken blocks**
  - Bookkeeper maintains same r_{AND}/r_{OR} as above
- **Whether an edge/block is ever executed** etc.
- **Path Frequencies**
  - Bookkeeper hashes value of ***r*** to counter-array

TFP – K-Persistent Paths

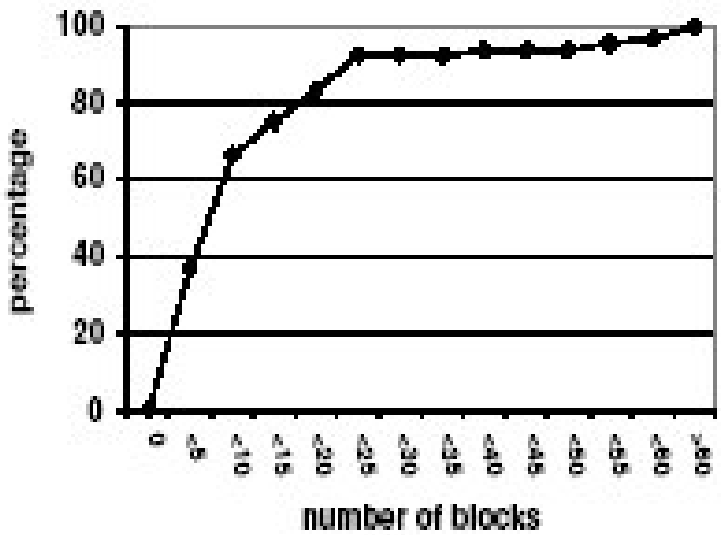


ACF	- 10010
ABEF	- 11001
ABDF	- 10101
ABCF	- 10011
ACEF	- 11010
ABCEF	- 11011

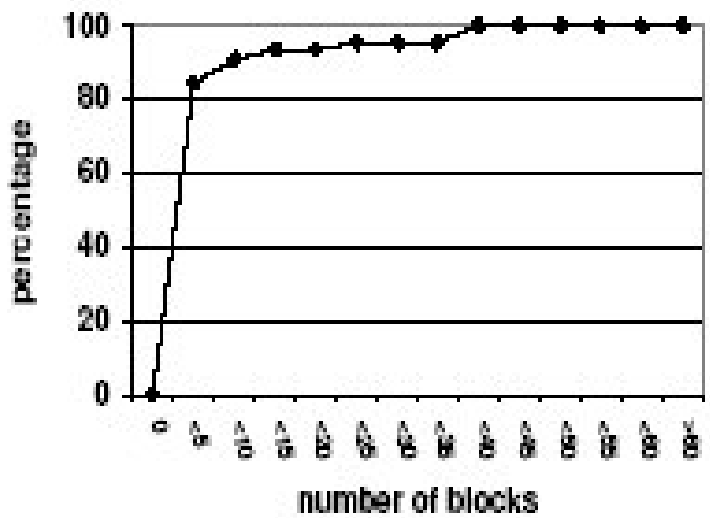
- Assume ABEF is always taken – $r_{AND} = 11001$, $r_{OR} = 11001$
- Had ACEF been taken too then $r_{AND} = 11000$ – implies EF always taken
- If all but ABDF is taken then $r_{OR} = 11011$ – implies that D is never taken

TFP – Salient Features

- Low overheads
  - Limited use of data structures – single variable for profiling, few for bookkeeeping
  - Profiles along blocks as opposed to edges
- Combines a range of benefits of Path/Edge/Block profiling in one framework
- Assumes that dense regions will be compact enough to profile using a few register(s)



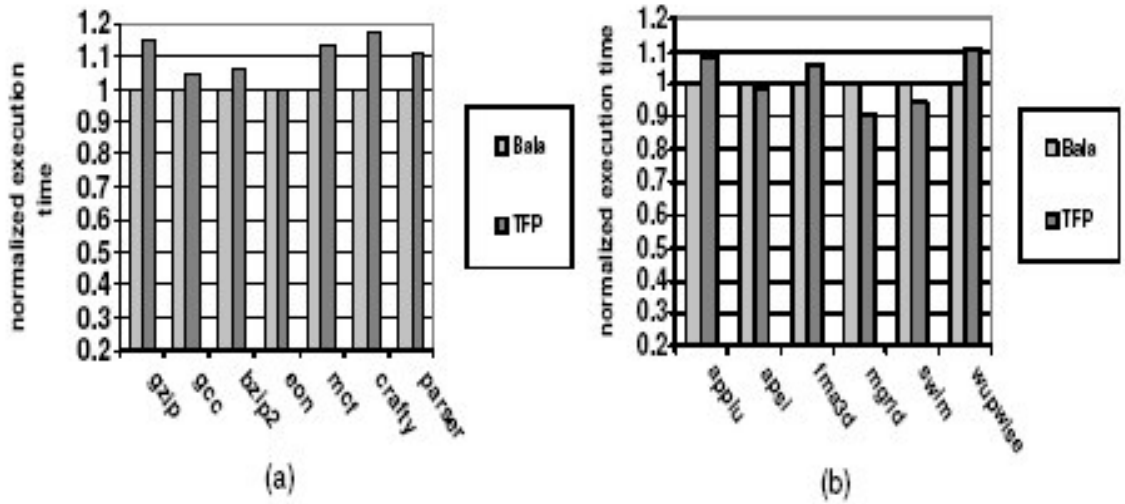
(a)



(b)

Experimental Results – Overheads of Using TFP

- Compared TFP against Bala's method of Path Profiling
  - Bala's method is known to run with near-zero overhead with compiler and architectural support
  - TFP should have lower overhead than Bala's method
  - We used ATOM which adds a large amount of overheads
  - TFP on an average is on an average 5.75% slower – can improve even further with optimizing compiler



Experimental results – Persistent Path Statistics

Benchmark Name	*Number of Static Paths Instrumented*	*Persistent Path Statistics*			
		K =50		*K =100*	
		paths	*%*	*paths*	*%*
cc1	33	11	98.14	10	96.67
gzip	3563	15	0.912	10	0.658
bzip2	1057	11	49.11	11	45.71
mcf	130	18	55.81	18	51.91
crafty	826	62	15.54	40	12.13
eon	20	4	3.109	3	3.108
parser	19623	40	45.90	35	41.18
AVG (INT)	**3607**	**23**	**38.36**	**18.14**	**35.91**
swim	9	4	99.99	4	99.99
applu	48	6	85.71	6	85.71
apsi	27	9	99.99	9	99.99
wupwise	18	4	61.54	4	61.54
mgrid	46	10	99.84	10	99.65
fma3d	94	16	75.05	16	75.05
AVG (FP)	**40.33**	**8.16**	**87.02**	**8.16**	**86.98**
AVG (net)	**1961**	**16.15**	**60.81**	**13.53**	**59.48**

Experimental results – Persistently Un-taken blocks

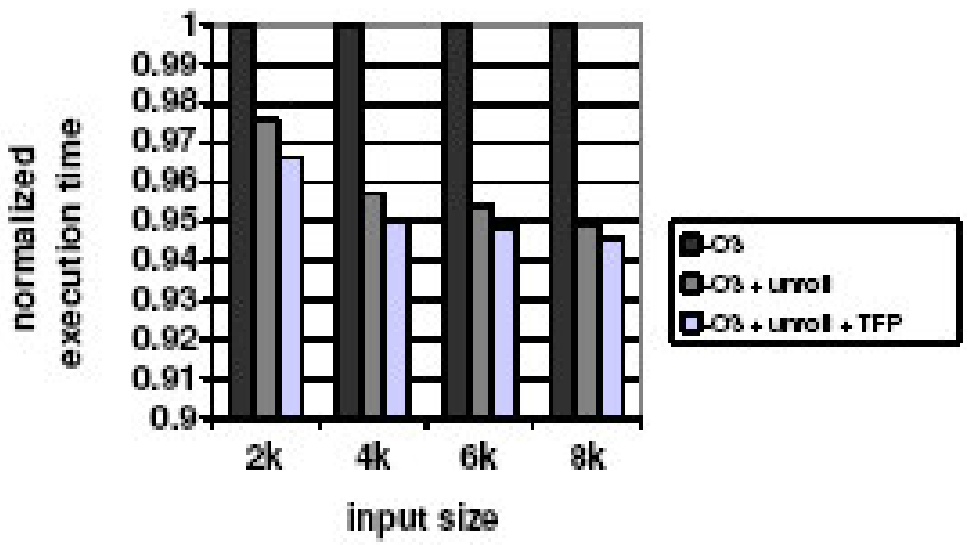
Benchmark Name	*Average Number of Blocks/instrumented region*	*% of Blocks NOT taken Persistently*	
		K =500	*K =1000*
cc1	6.6	38.97	31.80
gzip	16.4	22.94	21.73
bzip2	13.6	65.10	64.15
mcf	10.5	44.94	44.06
crafty	24.2	24.12	21.59
eon	10.0	0.017	0.017
parser	14.6	19.38	17.93
AVG (INT)	**14.27**	**30.78**	**28.76**
swim	4	0.000	0.000
applu	5	52.47	52.47
apsi	4.4	15.72	15.72
wupwise	7.4	39.18	39.18
mgrid	5.5	0.391	0.300
fma3d	12.5	54.22	54.18
AVG (FP)	**6.47**	**27.00**	**26.98**
AVG (net)	**10.67**	**29.03**	**27.94**

TFP and a Real Application

- Used TFP to perform a small optimization on RNA-fold

```
for(decomp = INFINITY; k=start_value; k<end_value;k++)
    if(decomp > Array1[k] + Array2[k+1][j])
        decomp = Array1[k] + Array2[k+1][j];
```

- Choose between multiple unrolled version of a loop based on persistence



Future Work and Conclusions

- Persistence should be involved for runtime profiling platforms
- TFP is an effective/efficient way to determine a range of Persistent Properties
- There exists dynamic Persistence in programs and should be utilized
- It is not clear what level of persistence is actually needed for optimizations
- Use TFP for security