Lazy Spilling for a Time-Predictable Stack Cache: Implementation and Analysis

Sahar Abbaspour[1] Alexander Jordan[1]
Florian Brandner[2]

Embedded Systems Engineering Section[1]
DTU Compute
Technical University of Denmark
{sabb,alejo}@dtu.dk

Computer Science and System Engineering Department[2]
ENSTA-ParisTech
florian.brandner@ensta-paristech.fr

Introduction

- Memory accesses are crucial for performance
  - Applies to time-predictable systems as well
  - Analyzable memory and cache designs
- Stack Cache
  - Specialized cache dedicated to stack data : Stack Cache
  - No need for precise knowledge about the addresses of accesses
  - Data transfers between the stack cache and the main memory

Stack Cache Background

- Two pointers: stack top (ST) and memory top (MT)
- Reserve: Stack frames are allocated upon entering a function: potential spilling
- Free: Stack frames are freed immediately before returning from a function
- Ensure: Compiler ensures a valid stack cache state: potential filling
- Initial conditions
  - `Initial pointers:` $ST = MT = 256$ `and` $|SC| = 8$
  - `Stack cache instructions are 2 words (8 bytes)`
  - `Invariant:` $|MT - ST| <= |SC|$

Stack Cache Example

```
function bar()
sres 2
sws [1] = ...
loop :
    lws ... = [1]
    call foo
    sens 2
    cmp ...
bt loop // jump to beginning of the loop
sfree // exit function
ret
```

`Invariant:` $|MT - ST| <= |SC|$

Reserve

$ST = 248$

Stack Cache Example

```
function bar()
sres 2
sws [1] = ...
loop :
    lws ... = [1]
    call foo
    sens 2
    cmp ...
bt loop // jump to beginning of the loop
sfree // exit function
ret
```

`Invariant:` $|MT - ST| <= |SC|$

Call

$ST = 216$ `and` $MT = 248$

Stack Cache Example

```
function bar()
sres 2
sws [1] = ...
loop :
    lws ... = [1]
    call foo
    sens 2
    cmp ...
bt loop // jump to beginning of the loop
sfree // exit function
ret
```

`Invariant:` $|MT - ST| <= |SC|$

Ensure

$MT = 256 \;\; ST = 216$

- $MT - ST = 40$: `Occupancy: 10 words`

Stack Cache Example

```
function bar()
sres 2
sws [1] = ...
loop :
    lws ... = [1]
    call foo
    sens 2
    cmp ...
bt loop // jump to beginning of the loop
sfree // exit function
ret
```

`Invariant:` $|MT - ST| <= |SC|$

Free

$ST = MT = 256$

Lazy Pointer

- Keeps track of the coherent elements of the stack cache and the main memory
- Invariant: $ST \leq LP \leq MT$
- LP divides the reserved space in the stack cache
  - Between ST and LP: *effective occupancy*
  - Between LP and MT: coherent data
- The effective occupancy area contains potentially modified data
- Store instruction writes to an address above the LP, some data potentially becomes incoherent

Implementation

Invariant: $ST \leq LP \leq MT$

Reserve

- Spills blocks between ST and LP
- No need to spill already spilled data: LP potentially moves down with MT
- MT and ST updated as in the original reserve

Free

- LP potentially moves down with ST (to satisfy the invariant)
- MT and ST updated as in the original free

Implementation

Ensure

- No change is needed

Store

- Potentially modifies the LP
  - Effective address of the store instruction
  - Current value of LP
  - LP might move up

Example

```
function bar()
sres 2
sws[1] = ...
loop :
  lws ...  = [1]
  call foo
  sens 2
  cmp ...
bt loop
sfree
ret
```

- Initial pointers: $LP = ST = MT = 256$ and $|SC| = 8$
- Stack cache instructions work on 2 words arguments (8 bytes)

Reserve

$LP = ST = 248$

Example

```
function bar()
sres 2
sws[1] = ...
loop :
  lws ...  = [1]
  call foo
  sens 2
  cmp ...
bt loop
sfree
ret
```

- Initial pointers: $LP = ST = MT = 256$ and $|SC| = 8$
- Stack cache instructions work on 2 words arguments (8 bytes)

Store

$LP = 252$

Example

```
function bar()
sres 2
sws[1] = ...
loop :
  lws ...  = [1]
  call foo
  sens 2
  cmp ...
bt loop
sfree
ret
```

- Initial pointers: $LP = ST = MT = 256$ and $|SC| = 8$
- Stack cache instructions work on 2 words arguments (8 bytes)

Call

- $MT = 248$
- Replacement of entire cache: $ST = LP = 216$

Example

```
function bar()
sres 2
sws[1] = ...
loop :
  lws ...  = [1]
  call foo
  sens 2
  cmp ...
bt loop
sfree
ret
```

- Initial pointers: $LP = ST = MT = 256$ and $|SC| = 8$
- Stack cache instructions work on 2 words arguments (8 bytes)

Ensure

- $MT = 256$: Occupancy 2 words
- $LP = ST = 248$: Effective Occupancy: 0 words

Static Analysis

Two Problems

- Worst-case filling of ensure instructions (Ensure Analysis)
  - Filling behavior of ensure instructions is not affected by lazy spilling
- Worst-case spilling of reserve instructions (Reserve Analysis)
  - Function-local data-flow analysis
    - Original SC: Utilized stack cache occupancy: $MT - ST$
    - Lazy spilling SC: Smaller region $LP - ST$: smaller effective occupancy
    - Store instructions
  - Inter-procedural analysis on the call graph

Static Analysis

- Simple data-flow analysis
- Computing worst-case effective occupancy before each calls
- Propagating maximum effective occupancy among instructions
- Computing context-sensitive bounds of the spilling for each function

Experiment Setup

- Patmos processor
- Mibench benchmarks
- Patmos LLVM compiler with full optimizations (-O3)
- Standard data cache with original stack cache and Lazy spilling stack cache

Experiments: Average Measurement

- Lazy Spilling Cache Performance
- Miss rate is not suitable for measurement
- $\frac{\#RD+\#WR}{\#Stalls}$

	SC_{128}			LP_{128}		SC_2			LP_2		DC
Benchmark	SC	DC	Spill	LP-SC	DC	SC	DC	Spill	LP-SC	DC	
basicmath-tiny	2.3	1.1	0.17	4.0	1.1	26.4	1.1	0.53	34.0	1.1	1.1
bitcnts	4.6	191.6	0.00	12.2	191.6	17054.7	193.7	0.71	19201.4	193.7	1.2
cjpeg-small	116.9	1.0	0.51	148.4	1.0	3470.7	1.0	0.09	6154.4	1.0	1.1
crc-32	9.0	0.9	0.03	21.3	0.9	814.9	0.9	1.00	814.9	0.9	0.9
csusan-small	11.3	2.2	0.16	18.6	2.2	1218.8	2.3	0.72	1430.0	2.3	1.5
dbf	477.4	1.0	0.47	623.0	1.0		1.0			1.0	1.0
dijkstra-small	19.5	1.4	0.20	32.8	1.4	335.2	1.4	0.54	433.7	1.4	1.4
djpeg-small	9.0	0.8	0.34	13.5	0.8	293.4	0.8	0.66	361.5	0.8	0.8
drijndael	15.8	0.9	0.20	28.7	0.9	185620.0	0.9	1.00	185620.0	0.9	0.9
ebf	172.5	1.0	0.44	224.6	1.0		1.0			1.0	1.0
erijndael	32.6	0.9	0.57	43.3	0.9	258340.0	0.9	1.00	258340.0	0.9	0.9
esusan-small	15.9	3.4	0.25	25.3	3.4	70.7	3.6	0.02	139.5	3.6	1.5
fft-tiny	3.1	1.1	0.08	5.8	1.1	85.0	1.1	0.56	103.4	1.1	1.1
ifft-tiny	3.1	1.2	0.08	5.9	1.2	83.1	1.1	0.56	101.0	1.1	1.1
patricia	2.5	1.0	0.27	4.2	1.0	26.4	1.0	0.55	31.9	1.0	1.0
qsort-small	3.1	1.0	0.62	3.7	1.0	7.8	1.0	0.76	8.6	1.0	1.0
rsynth-tiny	16.0	1.9	0.08	29.9	1.9	1096.1	1.9	0.48	1539.8	1.9	1.3
search-large	2.9	0.8	0.48	3.9	0.8	26.3	0.8	0.00	52.5	0.8	0.9
search-small	2.9	0.8	0.49	3.7	0.8	28.1	0.8	0.02	54.8	0.8	0.9
sha	8.3	1.6	0.20	14.1	1.6	668.7	1.6	0.91	700.6	1.6	1.6
ssusan-small	29.2	17.1	0.20	43.9	17.1	4313.5	17.1	0.80	4678.0	17.1	3.3

Experiments: Average Measurement

- Best result for LP_{128}: `bitcnts` with spilling totally avoided
- In the mean for LP_{256}: 30% reduction in spilling
- Worst results: `qsort-small` with 76% spilling
- Presence of a stack cache improves the performance of the data cache
  - `bitcnts` and `csusan`: Bytes per cycle of 1.2 jumps to 192.4 and 196.1 respectively
- Consistent improvements over all benchmarks

Experiments: Static Analysis

- Static worst-case compared to observations from dynamic execution
- Bytes spilled per stack cache context
- Maximum difference between statically predicted and observed spilling

	SC_{128} Max-Spilling-Δ			$LP\text{-}SC_{128}$ Max-Spilling-Δ		
Benchmark	Static	Dynamic	Gap	Static	Dynamic	Gap
basicmath-tiny	68,128	32,040	2.13×	10,052	8,080	1.24×
bitcnts	892	684	1.30×	768	320	2.40×
crc-32	844	652	1.29×	684	372	1.84×
csusan	5,404	2,592	2.08×	2,420	1,196	2.02×
dbf	684	456	1.50×	564	324	1.74×
dijkstra-small	10,220	5,796	1.76×	6,676	2,608	2.56×
drijndael	1,172	664	1.77×	1,024	488	2.10×
ebf	684	456	1.50×	564	324	1.74×
erijndael	880	400	2.20×	752	292	2.58×
esusan	4,724	1,888	2.50×	2,256	1,024	2.20×
fft-tiny	32,484	9,476	3.43×	5,804	3,712	1.56×
ifft-tiny	32,224	9,256	3.48×	5,620	3,548	1.58×
patricia	1,996	1,672	1.19×	1,804	984	1.83×
qsort-small	3,804	1,492	2.55×	2,432	840	2.90×
rsynth-tiny	109,864	15,320	7.17×	13,504	3,140	4.30×
search-large	840	740	1.14×	668	340	1.96×
search-small	828	728	1.14×	708	312	2.27×
sha	1,160	660	1.76×	1,032	448	2.30×
ssusan	6,608	1,824	3.62×	2,452	1,060	2.31×

Experiments: Static Analysis

- Static spill cost is reduced for all programs in the benchmark set
- When LP heavily reduces spilling, it also reduces the gap

	SC_{128} Max-Spilling-Δ			LP-SC_{128} Max-Spilling-Δ		
Benchmark	Static	Dynamic	Gap	Static	Dynamic	Gap
basicmath-tiny	68,128	32,040	2.13×	10,052	8,080	1.24×
bitcnts	892	684	1.30×	768	320	2.40×
crc-32	844	652	1.29×	684	372	1.84×
csusan	5,404	2,592	2.08×	2,420	1,196	2.02×
dbf	684	456	1.50×	564	324	1.74×
dijkstra-small	10,220	5,796	1.76×	6,676	2,608	2.56×
drijndael	1,172	664	1.77×	1,024	488	2.10×
ebf	684	456	1.50×	564	324	1.74×
erijndael	880	400	2.20×	752	292	2.58×
esusan	4,724	1,888	2.50×	2,256	1,024	2.20×
fft-tiny	32,484	9,476	3.43×	5,804	3,712	1.56×
ifft-tiny	32,224	9,256	3.48×	5,620	3,548	1.58×
patricia	1,996	1,672	1.19×	1,804	984	1.83×
qsort-small	3,804	1,492	2.55×	2,432	840	2.90×
rsynth-tiny	109,864	15,320	7.17×	13,504	3,140	4.30×
search-large	840	740	1.14×	668	340	1.96×
search-small	828	728	1.14×	708	312	2.27×
sha	1,160	660	1.76×	1,032	448	2.30×
ssusan	6,608	1,824	3.62×	2,452	1,060	2.31×

Summary

- Analyzable caches for time-predictable systems
- Stack cache
- Stack cache optimization
- Lazy spilling stack cache
- Static analysis of lazy spilling stack cache
- Results