SPECULATIVE PARALLELIZATION OF POINTER-BASED APPLICATIONS

Adrian Tineo,
Marcelo Cintra, and
Diego R. Llanos

Dpt. Computer Architecture, University of Malaga
School of Informatics, University of Edinburgh
Dpt. Computer Science, University of Valladolid

MOTIVATION:

- **Pointer-based applications** pose a challenge for dependence detection in current **parallelizing compilers**
- **Speculative parallelization** is useful for running in parallel code sections that:
  - cannot be fully analyzed by the compiler, or
  - exhibit a small amount of dependencies

KEY FEATURES:

- We extend an **all-software speculative parallelization engine** based on **sliding window** to support **pointer-based applications**
- We create a **scratch heap** for each window slot, which provides a working space for threads
- 3 tables provide the needed support for **scratch heap allocation**, **address translation** and **tracking possible conflicts** between accesses

OPTIMIZATIONS:

- We can choose **coarse- or fine-granularity** of heap locations (example is for fine-granularity)
- The speculative heap can be reduced with help from **compile-time analysis**:
  - pointer analysis
  - def-use chains analysis
  - shape analysis

Example program

```
// Pointer-based data structures
struct t1{
    int data;
    struct t1 * nxt;
}

struct t1 ** arr;

// Data structure creation
[...]

// Speculatively parallelized loop
for(i=0; i<4; i++){
    ptr = arr[i];
    val = ptr->data;
    ptr->data = val + 2;
    arr[i] = ptr->nxt;
}
```

Speculative parallelization engine with support for pointer-based programs
(version for 4 window slots and 2 processors)

- **At the start of the loop, a new scratch heap is created for every window slot, based on the current user heap, using the HAT.**
- **Each available thread takes the first free window slot and runs its block of statements.**
- **Scratch heaps are created empty. Values are fetched from the user heap as needed by load operations, using the HTT for translation.**
- **Heap accesses are recorded in the AST, with no need for memory fences. If a conflict arises, the offending thread is squashed.**
- **Non-speculative window slots that are done can be committed to the user heap, using the HTT and AST.**

Address	Size	Type
U(1)	4	ptr
U(2)	4	ptr
U(3)	4	ptr
U(4)	4	ptr
U(5)	2	non-ptr
U(6)	4	ptr
U(7)	2	non-ptr
U(8)	4	ptr
U(9)	2	non-ptr
U(10)	4	ptr
U(11)	2	non-ptr
U(12)	4	ptr

HAT (Heap Allocation Table): registers allocated pieces of memory in the heap, their size and type.

User heap	Type	Scratch heap 1	Scratch heap 2	Scratch heap 3	Scratch heap 4
U(1)	ptr	S1(1)	S2(1)	S3(1)	S4(1)
U(2)	ptr	S1(2)	S2(2)	S3(2)	S4(2)
U(3)	ptr	S1(3)	S2(3)	S3(3)	S4(3)
U(4)	ptr	S1(4)	S2(4)	S3(4)	S4(4)
U(5)	non-ptr	S1(5)	S2(5)	S3(5)	S4(5)
U(6)	ptr	S1(6)	S2(6)	S3(6)	S4(6)
U(7)	non-ptr	S1(7)	S2(7)	S3(7)	S4(7)
U(8)	ptr	S1(8)	S2(8)	S3(8)	S4(8)
U(9)	non-ptr	S1(9)	S2(9)	S3(9)	S4(9)
U(10)	ptr	S1(10)	S2(10)	S3(10)	S4(10)
U(11)	non-ptr	S1(11)	S2(11)	S3(11)	S4(11)
U(12)	ptr	S1(12)	S2(12)	S3(12)	S4(12)

HTT (Heap Translation Table): translates the user heap addresses to the different scratch heaps

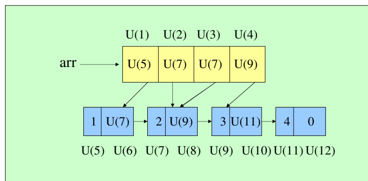
Window slot 1		Window slot 2		Window slot 3		Window slot 4	
Scratch heap 1	State	Scratch heap 2	State	Scratch heap 3	State	Scratch heap 4	State
S1(1)	**ExpLdMod**	S2(1)	NotAcc	S3(1)	NotAcc	S4(1)	NotAcc
S1(2)	NotAcc	**S2(2)**	**ExpLd**	S3(2)	NotAcc	S4(2)	NotAcc
S1(3)	NotAcc	S2(3)	NotAcc	**S3(3)**	**ExpLd**	S4(3)	NotAcc
S1(4)	NotAcc	S2(4)	NotAcc	S3(4)	NotAcc	S4(4)	NotAcc
S1(5)	**ExpLdMod**	S2(5)	NotAcc	S3(5)	NotAcc	S4(5)	NotAcc
S1(6)	**ExpLd**	S2(6)	NotAcc	S3(6)	NotAcc	S4(6)	NotAcc
S1(7)	NotAcc	**S2(7)**	**ExpLdMod**	**S3(7)**	**ExpLd**	S4(7)	NotAcc
S1(8)	NotAcc	S2(8)	NotAcc	S3(8)	NotAcc	S4(8)	NotAcc
S1(9)	NotAcc	S2(9)	NotAcc	S3(9)	NotAcc	S4(9)	NotAcc
S1(10)	NotAcc	S2(10)	NotAcc	S3(10)	NotAcc	S4(10)	NotAcc
S1(11)	NotAcc	S2(11)	NotAcc	S3(11)	NotAcc	S4(11)	NotAcc
S1(12)	NotAcc	S2(12)	NotAcc	S3(12)	NotAcc	S4(12)	NotAcc

AST (Access State Table): keeps track of accesses to scratch heaps. Possible states are: NotAcc, not accessed; ExpLd, read from user heap; Mod, modified; ExpLdMod, first read from user heap then modified

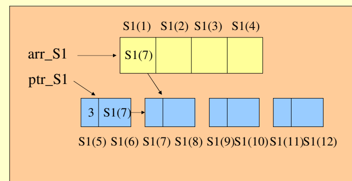
Thread 2, less speculative thread

Thread 1, more speculative thread

User heap

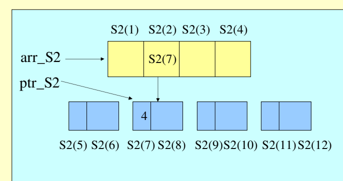


Window slot 1 Scratch heap 1



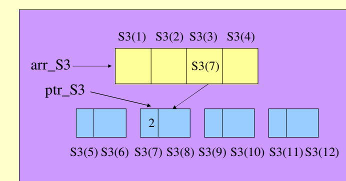
Window slot state: DONE

Window slot 2 Scratch heap 2



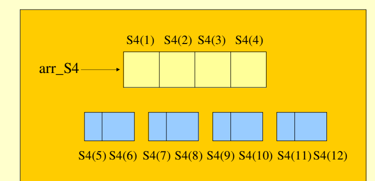
Window slot state: RUNNING

Window slot 3 Scratch heap 3

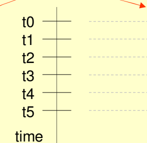


Window slot state: SQUASHED

Window slot 4 Scratch heap 4



Window slot state: FREE




```
ptr_S1 = arr_S1[0];
val_S1 = ptr_S1 -> data;
ptr_S1->data = val_S1 + 2;
arr_S1[0] = ptr_S1->nxt;
```

```
ptr_S2 = arr_S2[1];
val_S2 = ptr_S2 -> data;
ptr_S2->data = val_S2 + 2;
```

```
ptr_S3 = arr_S3[2];
val_S3 = ptr_S3 -> data;
```

Read-only pointers (such as arr) are translated to each scratch heap address space (arr_S1, arr_S2, ...). Variables which are first written then read (such as ptr or val), are made private to each window slot (ptr_S1, val_S1, ptr_S2, val_S2, ...)

Conflict: thread 1 reads S3(7) but S2(7) was modified in a less speculative window slot (both S3(7) and S2(7) refer to U(7)). Thread 1 must be squashed.