Pointer Safety and Data Races in UPC

Jean-Francois Collard, Alan H. Karp, Rob Schreiber
HP Laboratories Palo Alto
HPL-2003-179 (R.1)
December 16th , 2003*

E-mail: Jeff.Collard@hpl.hp.com, Alan.Karp@hp.com, Rob.Schreiber@hp.com

UPC, pointer safety, data race

The Unified Parallel C (UPC) language is a parallel extension of C that features both private and shared data that can both be accessed through pointers. Arithmetic on pointers to shared data is legal but as error-prone as standard C pointer arithmetic is. A consequence is that erroneous writes to shared data can occur, resulting in inadvertent data races even if the appropriate synchronizations coordinate the legitimate writes and reads. This paper makes several contributions to protect shared data from races in UPC programs, add safety checks on pointers-to-shared, and reduce the run-time overhead of these checks.

Pointer Safety and Data Races in UPC

Jean-Francois Collard, Alan H. Karp, Rob Schreiber

Jeff.Collard@hpl.hp.com Alan.Karp@hp.com Rob.Schreiber@hp.com

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

ABSTRACT

The Unified Parallel C (UPC) language is a parallel extension of C that features both private and shared data that can both be accessed through pointers. Arithmetic on pointers to shared data is legal but as error-prone as standard C pointer arithmetic is. A consequence is that erroneous writes to shared data can occur, resulting in inadvertent data races even if the appropriate synchronizations coordinate the legitimate writes and reads. This paper makes several contributions to protect shared data from races in UPC programs, add safety checks on pointers-to-shared, and reduce the run-time overhead of these checks.

Keywords

UPC, pointer safety, bounds checking, data race

1. INTRODUCTION

The Unified Parallel C (UPC) language is a parallel extension of C [1][2][3]. It provides a global address space, possibly on top of MPI or GasNET [15], and can therefore be ported to systems relying on message-passing as well as to machines with physically shared memory. At least three compilers are currently available: The HP product compiler [23], GWU's GNU compiler [18], and UC Berkeley's [13]. Because UPC is a SPMD language, each thread executes the same UPC code. Data are either private to a thread, or shared, in which case they are distributed across threads in a way specified by the user. Each thread has direct access (i.e., through standard C assignments and references) to its own private local data, to the local portion of shared data, and to remote shared data. (Note that, in this paper, the opposite of "shared" is "private" and the opposite of "remote" is "local.") Private and shared data can be accessed through pointers, and UPC extends C's pointer arithmetic to (possibly shared) pointers to shared data.

This feature is powerful, but can also be dangerous because it compounds well-known issues with pointers with the indeterminism of parallelism, such as data races. Data race detection has received renewed interest lately [7][8][21][22]. Usually, a data race involves a syntactically identified shared object, and an identified set of concurrent accesses, by identified threads, to that object. In contrast, a new dimension is added by UPC to this problem: allowing arithmetic on pointers-to-shared can potentially yield codes where a shared object that is not supposed to be accessed concurrently turns out to suffer a race — even though there may be no explicit write to that object. In other words, writes through errant pointers may interleave in unpredictable ways with legitimate writes to the same locations. As a consequence, while in most parallel languages data races result from incorrect synchronization, data races in UPC programs can occur *even if* synchronization primitives are correctly coordinating legitimate accesses to shared data.

One solution to avoiding such races would be to systematically acquire locks to shared variables. This would, however, incur a tremendous overhead. Instead, this work checks that there are no data races caused by pointers-to-shared arithmetic using techniques that have little overhead (and provably *none*, under some assumptions discussed in Section 7).

Also, we are interested in *undesired* data races. Therefore, we slightly change the definition of a data race, which is classically that two unsynchronized accesses refer to the same object and at least one access is a write. In addition, we require that *either both accesses are writes and they write distinct values, or the single write stores a value different from the value before the race.* For example, two unsynchronized assignments setting a shared scalar to the same value are not considered in a race. These writes are similar to silent stores [4].

We attack the problem on two fronts: On the one hand, we protect shared variables from incorrect accesses. On the other, we provide pragmas that let users request additional checks on their pointers, at minimal run-time costs. Specifically, we make the following contributions:

- We introduce a scheme that allows programmers to make sure the value of shared object hasn't been changed behind the back of a thread. Specifically, we detect some race conditions on specified objects in program parts that the programmer indicates as free of races on these objects. This detection produces no false positives, but may produce false negatives (i.e., fail to report data races) in corner cases. Experts in data race detection may find our scheme unsophisticated, but we believe most users will find it fits their needs in a simple, realistic and intuitive way.
- We introduce *localized* pointers-to-shared. For performance reasons, a pointer-to-shared is often intended to point to the local portion of a shared data structure. We make this intention explicit by adding a pragma, `localized`. Thanks to this information, the compiler can detect unintended misuses of pointers-to-shared.
- We extend to UPC pointers-to-shared known techniques to check that a pointer keeps pointing to its intended referent [5][6].
- We advocate that performing *some* run-time pointer checking, even if it is partial and/or late, is better than no checks at all. With this in mind, we disclose an opportunistic scheme for pointer checking that has provably zero run-time overhead.
- We present a low-overhead mechanism to perform bounds checking on processors of the Itanium architecture, even across calls to procedures compiled without support for bounds-checking.

Note that the last two points apply to standard C as well.

This paper is structured as follows. Section 2 provides a minimal introduction to UPC to make the paper self-contained. It also introduces our running example. Section 3 explains some formal properties of UPC. Section 4 exposes our mechanism to protect shared variables in selected code portions. Section 5 introduces user annotation to indicate the intended referent of a pointer-to-shared, and Section 6 introduces annotation to tell the compiler that a pointer-to-shared is meant to point to the local portion (as opposed to remote portions) of a shared object. Both annotations help the compiler assert properties on pointers-to-shared, those of Section 6 being of a novel type. Section 7 shows how to enforce part of the requested checks (possibly all, but also possibly none of them) within a given cycle budget, possibly 0, on processor architectures like the Itanium processor family.

2. UPC, AND A MOTIVATING EXAMPLE

UPC allows the number of threads to be defined either statically, or at program invocation time ("static" and "dynamic environment", in UPC terminology [3]). This number can still be explicitly referenced in the program text using the reserved keyword `THREADS`. Each thread has a unique integer identifier, `MYTHREAD`, whose value ranges between 0 and `THREADS`-1. Threads are created at program invocation only, i.e., UPC does not offer `parbegin/parend` constructs, nor does it provide equivalents of `fork()`, `join()`, or `run()`, making compiler analyses less complex in that respect than they are for languages such as Java or C extended with multithreading libraries.

Variable declarations have the usual C syntax, plus a new type qualifier, `shared`, and notations for allocation in blocks. For instance,

```
int shared i;
```

declares, reading from right to left, variable i, which is a shared integer. (Formally, the full type of `i` is "`int shared`", not just "`int`".) Again reading from right to left,

```
int shared * p;
```

declares p, a pointer to a shared integer. p itself is not qualified with `shared`, so it is a private pointer-to-shared. Finally,

```
int shared * shared sp;
```

declares a shared pointer to a shared integer. All threads can modify the value of both `sp` and `*sp`. Shared variables may not be automatic, but a private pointer-to-shared may.

Shared arrays may have multiple dimensions, but the size of exactly one dimension is a multiple of `THREADS` in the dynamic environment. (Exceptions to this rule are mentioned later.) Shared arrays can be spread across threads in a round-robin fashion, i.e., successive array elements are allocated on (have *affinity with*) successive threads. For instance, assuming two threads are used, the following declaration

```
double shared u[4*THREADS];
```

implies that `u[0]` has affinity with thread 0, `u[1]` has affinity with thread 1, `u[2]` with thread 0, etc. Elements of shared arrays can also be spread across threads in chunks:

```
double shared [B] v[..];
```

indicates that chunks of B (the *block factor*) elements should be allocated on threads in a round-robin fashion, starting by convention at thread 0. A block factor of 1 is equivalent to the default round-robin distribution. Observe the block factor itself is optional, meaning that

```
double shared [] w[THREADS];
```

is a valid declaration. It implies that all elements of `w` are allocated by convention on thread 0. A block factor of 0 has the same effect.

The compiler representation of a pointer-to-shared is a structure that contains three fields: `thread`, to indicate which thread the data pointed to has affinity with, `phase`, to indicate which element in a block the pointer points to, and `addr`, a local virtual address. Typically, the actual implementations of these fields are bit fields within a full, *global* virtual address (`thread` being encoded in the most significant bits, `addr` in the least bits, and `phase` in between). To access these fields, UPC features three primitives: `upc_threadof()`, `upc_phaseof()` and `upc_addrfield()`. To simplify the notation, however, and given a pointer-to-shared `p`, we denote these fields by `p.thread`, `p.phase` and `p.addr`, respectively.

Incrementing a pointer to a blocked shared array increments *both* the local address (by the size of the base type) *and* the phase, unless we are at the end of the block — in which case one of two scenarios occurs: "If the thread number is less than `THREADS-1`, the next increment increases the thread number by one and resets the local address to the beginning of the block and the phase to 0. If the thread number is `THREADS-1`, then the next increment resets the thread number to 0, increments the local address by the size of the base type so that is points to the beginning of the next block, and sets the phase to 0" [2].

Shared objects with affinity to a given thread can be accessed by that thread through either pointers-to-shared, or pointers-to-private after an appropriate cast. (If a pointer-to-shared is cast into a pointer-to-private on a thread that has no affinity with the shared object, the result is undefined.)

Of note, the language specification also requires local portions of shared arrays start at the same local address on all threads. Specifically, coming back to variable `u` declared earlier, and still assuming two threads, `u[0]` and `u[1]` have the same local address on their respective threads, and so do `u[2]` and `u[3]`, etc[1]. In other words, `(&u[0]).addr = (&u[1]).addr`, etc.

Shared memory can also be allocated dynamically. UPC provides three functions to that purpose: `upc_local_alloc(N,s)` allocates a chunk of memory of size N times s bytes in the shared portion of the caller thread, and makes that chunk accessible to all threads. The entire chunk has affinity with the thread that calls `upc_local_alloc()`. The two other functions, `upc_global_alloc(N,s)` and `upc_all_alloc(N,s)`, allocate N elements of s bytes across the shared parts of all threads' memories. (Typically, N is a multiple of `THREADS`.) They differ in that the former is called by one thread, and the latter by all. Elements are allocated cyclically, starting with thread 0, and local portions of the allocated space start at the same local addresses. The pointer returned by `upc_global_alloc()` or `upc_all_alloc()` is the same on all threads and points to the first allocated element, which sits on thread 0.

On the control-structure side, UPC features classical synchronization primitives, like `barrier`. It also introduces `upc_forall`, which describes a global loop that maps, at compile- or at run-time, loop iterations to threads. The syntax of

[1] It doesn't mean the *global* addresses of these objects are the same. In particular, this makes UPC suitable for shared-memory multiprocessors.

upc_forall is that of C's for, plus a fourth argument that specifies the affinity. For instance, assuming the previous declaration of u and execution on two threads, the following construct:

```
  upc_forall(i=0; i<U; i++; &u[i]) {
     u[i]++;
  }
```

requests thread 0 to increment all even elements of u by 1, and thread 1 to do the same on all odd elements. In general, for an arbitrary shared array t of at least U elements, the loop:

```
 upc_forall(i=0; i<U; i++; &t[i]){ ... }
```

is equivalent, in its function but not necessarily in its implementation, to:

```
  for(i=0; i<U; i++) {
    if (upc_threadof(&t[i]) == MYTHREAD) {
      ...
    }
  }
```

The other version of upc_forall is:

```
 upc_forall(i=0; i<U; i++; f(i)){
   ...
 }
```

where f(i) is an integer expression. It is functionally equivalent to:

```
  for(i=0; i<U; i++) {
    if (f(i) == MYTHREAD) {
      ...
    }
  }
```

We can now introduce our running UPC example, below. Lines 1, 2 and 3 declare three shared arrays, p being an array of pointers. Recall that shared objects are global. Notice that all three arrays have the same affinities with threads. To make our point, we make elements of t of the same size as that of pointers on the target platform, which in our case is "LP64" i.e., longs and pointers are 64-bit long. (That is, the type of t could have been __int64 or double). Lines 5 and 6 declare locals, including private pointer-to-shared t2. Lines 7 through 11 initialize u, t, and p. Some values get printed on lines 12—14 for the sake of the discussion. Please note that these printf statements are here for illustration purposes only and are ***not*** correct UPC code: UPC pointers should not be printed directly, since that output is implementation dependent. (Correct UPC programming would print the upc_threadof(), upc_phaseof(), and upc_addrfield() fields separately.) Line 15 sets t2 to the address of the element of t that is local to (has affinity with) the current thread. Line 16 increments t2 and therefore makes t2 point to a remote (non-local) element of t, a point elaborated upon in Section 6. Line 18 assigns 0 to presumably the element of t pointed to by t2, but as explained later, its execution on thread THREADS -1 in fact corrupts p[0], which is dereferenced by thread 0 on line 20.

```
1  long shared t[THREADS];
2  int shared * shared p[THREADS];
3  int shared u[THREADS];
4  void main() {
5    int i;
6    long shared * t2;
7    upc_forall(i=0;i<THREADS; i++; &t[i]){
8      t[i] = 1;
9      u[i] = 2;
10     p[i] = &u[i];
11   }
12   printf("Thread %d: &p[%d]=%p\n",  MYTHREAD, MYTHREAD, &p[MYTHREAD]);
13   printf("Thread %d:  p[%d]=%p\n",  MYTHREAD, MYTHREAD, p[MYTHREAD]);
14   printf("Thread %d: *p[%d]=%d\n",  MYTHREAD,MYTHREAD, *p[MYTHREAD]);
15   t2 = &t[MYTHREAD];
16   t2++;
17   printf("Thread %d: t2=%p\n",  MYTHREAD, t2);
18   *t2 = 0;
19   printf("Thread %d:  p[%d]=%p\n",  MYTHREAD, MYTHREAD, p[MYTHREAD]);
20   printf("Thread %d: *p[%d]=%d\n",  MYTHREAD,MYTHREAD,*p[MYTHREAD]);
21   printf("Thread %d completed OK\n",  MYTHREAD);
22 }
```

Let us now run this program on two threads. When thread 1 executes the statement at line 16, the index expression goes out of the bounds on `t`. Due to the way we declared the shared variables, it might be that `p` is allocated in memory right after `t` — and so it is on the platform[2] we've been experimenting with. Because the type pointed to by `t2` has the same size as pointers do, all 64 bits of `p[0]` get set to 0. The output of the execution then looks as follows. The `prun` command allows specifying the number of threads at launch time (here, 2).

```
> prun -n 2 a.out

Thread 0: &p[0]=1400002f8
Thread 0:  p[0]=140000300
Thread 0: *p[0]=2
Thread 0: t2=200001400002f0
Thread 0:  p[0]=0
Thread 1: &p[1]=200001400002f8
```

[2] This experiment was done using the HP UPC compiler and run on an AlphaCluster.

```
Thread 1:  p[1]=20000140000300
Thread 1: *p[1]=2
Thread 1: t2=1400002f8
Thread 1:  p[1]=20000140000300
Thread 1: *p[1]=2
Thread 1 completed OK
prun: ./a.out (pid 829826) killed by signal 11 (SIGSEGV)
```

We can observe that the addresses of `p[0]` and `p[1]` have the same low bits on their respective threads, i.e., their local addresses are equal. After the increment of `t2` at line 16, the resulting global address is printed, and we can check its value on thread 1 equals that of `p[0]` on thread 0. Therefore, the assignment at line 18 incorrectly sets `p[0]` to 0, which is witnessed by the last message printed by thread 0. In conclusion, thread 1 corrupts the portion of shared data that is local to thread 0. Thread 0 crashes, but thread 1 completes OK.

Observe also that the bug comes from manipulation of pointers related to `t`, but an unrelated data structure gets corrupted: There is no algorithmic connection between `t` and `p`.

3. SOME PROPERTIES

3.1 Properties of Array Layouts

Let us consider an array `t` declared by:

```
    type shared [B] t[N];
```

First, recall that by convention in C, the identifier `t` equals the address of `t[0]`. Let us consider the case B=0 first, which we saw means all elements of `t` are allocated in thread 0's local shared space. Because all elements are in the same space, the `addr` field does not need to be considered yet. For any element `t[x]` of `t`, we have:

$$\forall x,\ 0 \le x < N:\quad t \le \&t[x] \le t + (N-1) * \mathrm{sizeof}(\mathit{type})$$

For B>0, an element `t[x]` has affinity with the thread whose index is

$$(\&t[x]).\mathtt{thread} = (x \text{ div } B) \bmod \mathtt{THREADS}$$

where mod denotes the integer modulo operator. Let K=N div B be the number of full chunks of B elements spread across the threads. (div denoting the integer division operator.) In general, only bounds on the `addr` field can be derived. Clearly, the lower bound is tight:

$$\forall x,\ 0 \le x < N:\quad (\&t[x]).\mathtt{addr} \ge t.\mathtt{addr} \tag{1}$$

Regarding upper bounds:

- If K is not a multiple of `THREADS`, then at least one thread stores ceiling(K / `THREADS`) chunks. This is illustrated in Figure 1. Thus:

$$(\&t[x]).\mathtt{addr} \le t.\mathtt{addr} + (\mathrm{ceiling}(K / \mathtt{THREADS}) * B - 1) * \mathrm{sizeof}(\mathit{type}) \tag{2}$$

Clearly, more precise thread-dependent bounds could also be derived at the price of potentially higher overhead in the corresponding code.

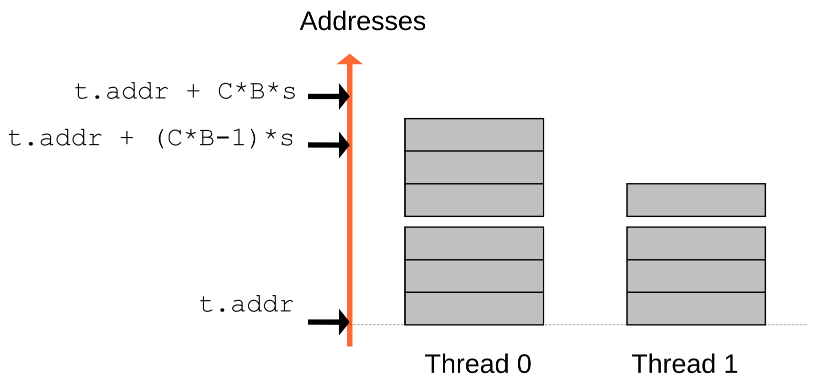


Figure 1: Array t of 10 elements of size s is spread across two threads by blocks of B=3. K equals 3, so they are three full chunks of B elements. C denotes ceiling(K/THREADS), which equals 2.

- If K is a multiple of THREADS, then each thread stores K/`THREADS` chunks of B elements of `t`, and thread 0 stores the remaining N-KB elements (there are at most B-1 of them). This is illustrated in Figure 2. An upper bound is therefore:

$$\texttt{(\&t[x]).addr} \leq \texttt{t.addr} + (N - KB + KB/\texttt{THREADS} - 1) * \text{sizeof}(\textit{type}) \tag{3}$$

Equations (2) and (3) can be readily extended to multidimensional arrays as long as one array dimension is spread across processors.

3.2 Properties of the Layout of Dynamically Allocated Data

Consider the following declaration and definition:

```
int shared * a = upc_local_alloc(N,s);
```

All N elements are allocated on the calling thread, and therefore bounds can hold on the full pointers. (In turn, they imply inequalities on the local addresses, i.e. on the `addr` fields.) A pointer-to-shared `p` to a piece of this data is such that:

$$\texttt{a} \leq \texttt{p} \leq \texttt{a + (N-1) * s} \tag{4}$$

However, when dynamically allocated memory is spread across threads, only bounds on the `addr` field can be derived. For either allocation primitive,

```
int shared * a = upc_global_alloc(N,s);
```

or

```
int shared * a = upc_all_alloc(N,s);
```

bounds on the `addr` field of a pointer `p` to these data are:

$$\texttt{a.addr} \leq \texttt{p.addr} \leq \texttt{a.addr} + (\text{ceiling}(N / \texttt{THREADS}) - 1) * s \tag{5}$$

If N is a multiple of THREADS, the upper bound on `p.addr` is simply `a.addr` + (N/THREADS-1)* s, and is the same of all threads. (Again, this is the case in the commonly used dynamic environment.) If it isn't, thread-dependent bounds more precise than (5) could also be derived.

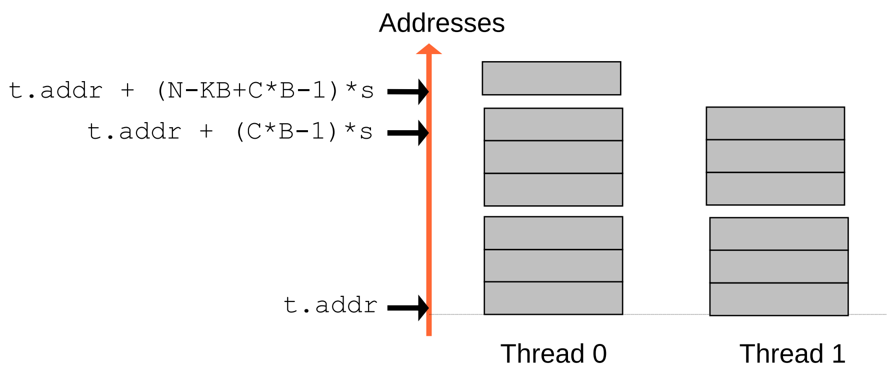


Figure 2: Case where K is a multiple of THREADS. Array t of N=13 elements of size s is spread across two threads by blocks of B=3. K equals 4, so each thread has two full chunks of B elements. C denotes ceiling(K/THREADS)=K/THREADS, which equals 2. Thread 0 gets the remaining N-KB=1 element.

3.3 Properties of Loops

Let us consider the first form of the `upc_forall`:

```
 upc_forall(i=0; i<U; i++; &t[i]){ ... }
```

Also, we restrict the subscript of `t` to be the loop counter (but note that nearly all `upc_forall` loops of the first form satisfy this constraint). The iteration subset that thread `MYTHREAD` must execute is, by definition:

$$S(\texttt{MYTHREAD})=\{i:\ i\in\mathbf{N},\ \ 0\le i<U,\ \ \texttt{upc\_threadof(\&t[i])}=\texttt{MYTHREAD}\ \}$$

where **N** denotes the set of natural numbers and comma denotes conjunction. Equivalently:

$$S(\texttt{MYTHREAD})=\{\ i: i\in\mathbf{N},\ \ 0\le i<U,\ \ (i \text{ div } B) \bmod \texttt{THREADS}=\texttt{MYTHREAD}\ \}$$

- If the block factor B of `t` is 1, then this set is:

$$S(\texttt{MYTHREAD})=\{\ i:\ \exists k\in\mathbf{N},\ k\ge 0,\ \ \ i=\texttt{MYTHREAD}+k\ \texttt{THREADS},\ \ i<U\ \}$$

We have:

$$0\le k<(U-\texttt{MYTHREAD})\ /\ \texttt{THREADS}$$

and since k is an integer, this is equivalent to:

$$0\le k<\text{ceiling}((U-\texttt{MYTHREAD})\ /\ \texttt{THREADS})$$

Therefore, the element count of S(`MYTHREAD`) is:

$$| S(\texttt{MYTHREAD}) | = \text{ceiling}\,((U - \texttt{MYTHREAD})\,/\,\texttt{THREADS})$$

- More generally, for any $B > 0$, the iteration subset is:

$$S(\texttt{MYTHREAD}) = \{\, i : \ 0 \le i < U, \quad \exists k \in \mathbf{N},\ k \ge 0,\ (i \text{ div } B) = \texttt{MYTHREAD} + k\ \texttt{THREADS} \,\}$$

Let $j = i$ div B. By definition, $\exists q \in \mathbf{N}$, $0 \le q < B$, $i = B\,j + q$. Thus:

$$S(\texttt{MYTHREAD}) = \{\, i : \ \exists j,k,q \in \mathbf{N},\ k \ge 0,\ i < U,\ \ j = \texttt{MYTHREAD} + k\ \texttt{THREADS}, \quad 0 \le q < B,\ i = B\,j + q \,\}$$

To count the number of elements in this set, let us first consider the set of j's that satisfy the above constraints. This set is:

$$\{\, j : \ j \in \mathbf{N},\ \ \texttt{MYTHREAD} \le j < U/B, \quad (\exists k \in \mathbf{N}, k \ge 0, j = \texttt{MYTHREAD} + k\ \texttt{THREADS}) \,\}$$

The number of element in that set is ceiling(((U/B)- MYTHREAD) / THREADS)

For each j in that set, there are B values of i except perhaps for the last (i.e., maximal) value of j. Therefore, a (close) upper bound for the element count in S(MYTHREAD)

$$| S(\texttt{MYTHREAD}) | \le \ B * \text{ceiling}(((U/B) - \texttt{MYTHREAD})/\texttt{THREADS})$$

4. PROTECTING SHARED OBJECTS

By protecting shared objects, we mean to ensure that a thread has temporary but exclusive write access to a shared object over a piece of code specified by the user. We introduce a compiler directive or pragma, `protect`, which the compiler translates into run-time assertions. This pragma is just syntactic sugar for code that the programmers could write themselves; A consequence is that the pragma has an unambiguous semantics (more precisely, its semantics is as unambiguous as its C translation is), but has the same limitations as user-level C code has (e.g., uses no kernel-mode system call, no hardware-specific assembly instructions, etc.)

The pragma applies to an explicit scope, which follows the pragma immediately. The compiler inserts an assertion before each access to a shared object, and a store operation after each update. The store records in a thread-local variable the value the thread expects the object to have when next encountered, and the test makes sure the object has the expected value. An error message can be generated when a discrepancy is detected. Several threads can protect the same object.

An excerpt of a thread is shown below. On the left is the original code. On the right is the code after applying protection. Suspension points indicate irrelevant pieces of code. Within the protected region, suspension points contain no reference to `x`.

```
1  int a;                          1  int a;
2  int shared x=3;                 2  int shared x=3;
3  ...                             3  ...
4  #pragma protect x
4b {                               4b {
                                   4c   int saved_x = x;
5    ...                           5    ...
6    a = x;                        6b   if(x != saved_x)
```

```
                                 6c      error();
                                 6d   a = saved_x;
7    ...                         7    ...
8    x = foo;                    8b   if(x != saved_x)
                                 8c       error();
                                 8d   tmp = foo;
                                 8e   x=saved_x = tmp;
9    ...                         9    ...
                                 9b   if(x != saved_x)
                                 9c       error();
10 }                             10 }
```

On line 6d, assigning the `saved_x` to `a` instead of `x` assures us that we're using the expected value even if another thread modifies `x` between lines 6b and 6d. The same care is needed when modifying `x`, on lines 8b—8e. To see why, imagine two threads increment `x`. Each thread executes three instructions: read `x`, add 1, and write `x`. The classical problem here is that any interleaving of the six instructions is possible, and in particular one thread may have modified `x` after the other has read `x` but before it has updated it. In the absence of atomic increments, each thread wants in general to make sure the value of a protected object hasn't changed behind its back even when it is about to write into it. Hence the need for lines 8b and 8c. Such a detrimental interleaving can occur anywhere within the scope, so we systematically add a final check on exit from the scope (in this example, at lines 9b and 9c). This final check has insignificant cost in the case of scalars; in the case of arrays (discussed later) it requires a loop over all protected elements, a potentially hefty overhead. It catches most but not all cases, as discussed later.

Several observations are in order here:

- Our scheme does not completely say *where* the race occurs, i.e. it identifies only one access in the pair of accesses.
- Our scheme only protects syntactical accesses to a shared object, i.e., a reference must explicitly contain the name of the object, as opposed to accessing the object through a pointer. For instance, a consequence is that updating x through a pointer on line 8, instead of a syntactical write to x, is illegal in a protected scope:

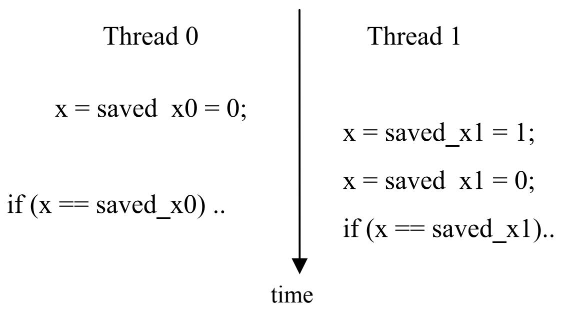
  ```
  8   int *p=&x;  *p = foo;
  ```

  Since we assume no pointer information, the pragma preprocessor would fail to see that line 8 is an update of x, resulting in an execution-time error later in the protection scope.
- Finally, observe as well that protection is free of false positives: Any reported error does correspond to a change of value of a protected object. It also makes the implementation simple: In particular, we do not need to maintain a history of accesses to each protected object, in contrast to e.g. [7] and [25].
- The alert reader may have noticed that the value of x at lines 6b, 8b or 9b may be stale. Specifically, with relaxed references, a write-through cache, and a cached copy of `x`, the reads of `x` may well return the same value as the one saved in the `saved_x` variable. However, this is missing the point: Under the same conditions, the original program

would have not seen the update of x. We have no reason to require the protection mechanism to observe more than the original program did.

False negatives can still happen, however, in two situations:

- One is when two successive write-write races to the same object, the same thread wins both but resets the shared variable to its original value. This is illustrated on the figure below, where the vertical axis represents time.



  The first data race is won by thread 1, which sets shared variable X to 1. At this point, the values of saved_x0 on thread 0 and that of x do not match, but no test is performed yet. Before such a test can be executed, the second race resets x to 0. The value of x has changed twice behind the back of thread 0, but our mechanism did not catch it.

- The other situation is when an object is modified on exit of the protection scope, just after it was checked one last time. Consider the following interleaving on exit from the protected scope:

```
// On Thread 0:                          // On Thread 1:
 9b   if(x != saved_x0)
                                          8e   x = saved_x1 = tmp1;
                                          9b   if(x != saved_x1)
 9c        error();                       9c        error();
 10   }                                   10   }
```

  Neither thread 0 nor thread 1 detects the race, and a subsequent read of x may fetch a non-deterministic value. Note that this can only happen on exit from a protected scope (otherwise, the check on exit of the scope will catch the race.) However, the solution is simple: The scope should be extended to include, in particular, that subsequent read.

  Similarly, when array elements are checked on exit from a protection scope, it may be that the first element, after it is checked one last time, suffers a race by the time the checking loop completes. The solution is identical.

Notice that protection is different from acquiring a lock. Functionally, it does not guarantee correctness, since it only reports an error if a race occurs. On the positive side, it is also less expensive. Note also that this protection mechanism works whether the strict or relaxed consistency model is adhered to[3].

[3] Shared data accessed with UPC's relaxed model can be written and read without cross-thread synchronization. Both models can coexist in the same program — but must of course apply to different data.

The back-up variable (in the previous example, `saved_x`) is declared in the explicit scope. We also require that there are no incoming control-flow edges into the protection scope from outside the scope. The reason for this is that the code that initializes the back-up object may not otherwise be executed. In other words, we want the initialization code of the back-up variable to dominate [16] any use of that variable. There may be outgoing edges, without risk of memory leaks.

To make sure back-up data structures can be statically allocated, the directive is restricted as follows:

```
#pragma protect object
{
   // Scope where the pragma applies
}
```

where ***`object`*** is either:

- the name of a scalar, in which case the scalar is protected by all threads; or
- the name of an array, in which case the entire array is protected by all threads; or
- an array element identified by compile-time constants; this array element is protected by all threads; or
- the name of an array followed by two brackets, like:

  ```
  #pragma protect p[]
  ```

  This tells the compiler to protect on each thread but within the following C scope all elements of `p` that have affinity with the thread. The block factor of `p` must be greater than 0.

- the name of an array followed by two brackets surrounding the name of another array, like:

  ```
  #pragma protect p[t]
  ```

  Array `t` must be of the same shape as `p`, i.e., they must have the same dimension and sizes of each dimension must be equal. The block factor of `t` must be greater than 0. The pragma requests the compiler to protect the elements of `p` whose corresponding elements of `t` have affinity with the current thread.

The code the compiler generates for each form of the directive is explained in the next two sections.

4.1 Compiler Action on Entry to a Protected Scope

When presented with `protect`, the questions the compiler has to solve are:

1. What is the size of the local array each thread will maintain to back up the elements of `p` it protects — so that the code to declare the local array can be generated on entry to the protection scope?
2. What code should be generated to initialize the back-up array after its declaration?

If the protected object is a scalar, an array element, or an entire array, the answers are obvious. We consider the remaining two forms of the pragma in the rest of this section. Let's assume the rank of `p` is 1, and let N be the size of this dimension.

First, keep in mind that the language definition requires that we are either in a static environment, in which case N and `THREADS` are compile-time constants, or in a dynamic environment, in which case N is guaranteed to be a multiple of `THREADS`.

The first problem is to generate the declaration of `saved_p`, the private array that backs up array `p`:

```
    type saved_p[size];
```

where ***`type`*** is the type of p. Let B>0 be the blocking factor of `p` or `t`, depending on the form of pragma we consider. We use results of section 3.1:

- If B equals 1, ***`size`*** = `N/THREADS`.
- If B is greater than 1, remember that thread `MYTHREAD` protects as many elements of p as the element count in this subset, which we saw has the following upper bound:

  B * ceiling(((N/B)- `MYTHREAD`) / `THREADS`)

  The static size of the local back-up array must be the maximum of that bound over all threads, and therefore:

  `size` `=` B * ceiling((N/B) / `THREADS`)

The second problem is to initialize `saved_p`, which is equivalent to compiling the following loop:

```
int k = 0;
upc_forall(i=0; i<N; i++; expr){
    saved_p[k++] = p[i];
}
```

where ***`expr`*** is either `&t[i]` or `&p[i]`, depending on which pragma form we consider.

- If B equals 1, the pragma makes the compiler generate the following code:

  ```
  int k=0;
  for (i=MYTHREAD; i<N; i+=THREADS) {
     saved_p[k++] = p[i];
  }
  ```

- If B is greater than 1, the pragma makes the compiler generate code equivalent to the following:

  ```
  int k=0;
  for (j=MYTHREAD; j<N/B; j += THREADS){
    for (q=0; q<min(B,N-(j*B)); q++) {
       i = j*B + q;
       saved_p[k++] = p[i];
    }
  }
  ```

4.2 Compiler Action on a Reference to a Protected Object

The question to be answered now is: Given an arbitrary element `p[x]`, how does the compiler know (a) if `p[x]` is protected in the current thread (and therefore decide if it has to generate protection code), and (b) what the index of `saved_p` that backs `p[x]` up is? Again, if the protected object is a scalar, an array element, or an entire array, the answers are obvious. For the remaining two forms, answering (a) boils down to checking if `p[x]` (or `t[x]`, depending on the pragma form) has affinity with `MYTHREAD`. Answering (b) however depends again on the block factor B of either `p` or `t`, depending on which form of the pragma is relevant:

- If B = 1, answering the question is equivalent to solving:

  $$\exists k \geq 0 \text{ s.t. } \texttt{MYTHREAD} + k\,\texttt{THREADS} < N,\ \texttt{MYTHREAD} + k\,\texttt{THREADS} = x$$

  Given any reference `p[x]`, the index k of the element of the local array that backs `p[x]` up is:

  $$k = (x - \texttt{MYTHREAD}) / \texttt{THREADS}$$

  The compiler should check that (a) k is a nonnegative integer and (b) `MYTHREAD` + k `THREADS` < N. If it can't check both conditions statically, it should generate code to that purpose. If both conditions hold, then `p[x]` is backed up by `saved_p[k]`. Otherwise, `p[x]` is not protected by `MYTHREAD`.

- If B > 1, the question is equivalent to solving:

  $$\exists j,k,q \in \mathbf{N} \text{ s.t. } k \geq 0,\quad j = \texttt{MYTHREAD} + k\,\texttt{THREADS},\quad 0 \leq q < B,\quad x = B\,j + q,\quad x < N$$

Notice that there is at most one pair (k,q) satisfying this equation, which can be found by computing `q = x mod B` and `j = x div B`, and then checking that `j≥MYTHREAD.` `k` is then equal to `(j-MYTHREAD)/THREADS`. If there is a solution (k,q), then `p[x]` is backed up by `saved_p[B*k + q]`.

4.3 Compiler Action on Exit from a Protected Scope

On exit from a protected scope, all protected elements are checked against their respective back-up variables, in any order. For the last two forms of the pragma, the corresponding code is equivalent to:

```
k=0;
upc_forall(i=0; i<N; i++; expr){
  if (saved_p[k++] != p[i])  error();
}
```

where *expr* is either `&t[i]` or `&p[i]`, depending on which form we consider.

4.4 Back to the Example

To protect the elements of `p` in the running example, we insert Statements 6b, 6c and 21b below (some lines are omitted):

```
4   void main() {
5     int i;
6     long shared * t2;
6b  #pragma protect p[]
```

```
6c  {
7     upc_forall(i=0;i<THREADS;i++;&t[i]){
8       t[i] = 1;
9       u[i] = 2;
10      p[i] = &u[i];
11    }
...
15    t2 = &t[MYTHREAD];
16    t2++;
17    printf("Thread %d: t2=%p\n", MYTHREAD, t2);
18    *t2 = 0;
...
21b }
```

In the example, N/THREADS = THREADS/THREADS = 1, thus a single element is (statically) allocated in `saved_p`. The translation of the annotated source code begins as follows:

```
6c {
6d   int shared * saved_p[1];
6e   int k=0;
6f   for(i=MYTHREAD; i<THREADS; i+=THREADS)
6g      saved_p[k++]= p[i];

7    upc_forall(i=0;i<THREADS; i++; &t[i]){
7b     int shared * tmp;
8      t[i] = 1;
9      u[i] = 2;
9c     tmp = &u[i];
       //Subscript is strengh-reduced to 0:
9d     saved_p[(i-MYTHREAD)/THREADS]=tmp;
10     p[i] = tmp;
11   }
```

In addition, each reference to an element of `p` is preceded by code to check its protection. Statements at lines 13, 14, 19 and 20 contain such a reference[4] and therefore are preceded by such code. For instance, line 19 is preceded by:

```
18b  if (saved_p[0] != p[MYTHREAD]) {
18c    error();
18d  }
```

[4] The statement on line 12 does not contain a reference to an element of p; it only takes the *address* of `p[MYTHREAD]`.

For reasons discussed earlier, line 19 itself is changed so that the saved value is used instead of `p[MYTHREAD]`:

```
19   printf("Thread %d:  p[%d]=%p\n", MYTHREAD, MYTHREAD, saved_p[0]);
```

Notice that lines 6e through 6g are the brute-force application of the discussion in Section 4.1 — but a smart compiler can replace the three of them with

```
6e        saved_p[0]= p[MYTHREAD];
```

Line 9d is also the brute-force application of the discussion in Section 4.2. In the example, the `upc_forall` loop spawns one iteration per thread, and that iteration is such that i equals `MYTHREAD`. Therefore, the subscript on line 9d can statically be strength-reduced to 0, which satisfies conditions (a) and (b) of case B=1.

The same rules are applied to compute the subscript of `saved_p` on line 18b. Unknown k equals `(MYTHREAD - MYTHREAD)/THREADS`, which equals 0; It is a nonnegative integer, and on all thread, we clearly have `MYTHREAD <` N since N equals `THREADS`. Therefore, the back-up storage for `p[MYTHREAD]` is indeed `saved_p[0]`.

5. INTENDED REFERENTS

One way to address pointer errors is to add safeguards to their use. In the running example, the bug clearly comes from the out-of-bound increment of `t2` on line 16 and the dereference of `t2` on line 18. This can be detected by bounds checking, assuming we know what bounds to check.

Patil and Fischer [5] define the intended referent as the object a pointer is meant to reference. This is indeed a very intuitive concept assuming, as they do, that the program does not cast non-pointers into pointers.

We extend this concept to pointers to shared objects, and introduce pragma `sticky` to make the referent explicit. Its syntax is either

```
    #pragma sticky p : ptr
    {
       // Scope where the pragma applies
    }
```

or:

```
    #pragma sticky p : ptr : X
    {
       // Scope where the pragma applies
    }
```

where *`p`* and *`ptr`* are pointers-to-shared[5] and *`X`* is an integer expression. *`p`* and *`ptr`* must point to the same type *`type`*. The intended referent of *`p`* is the object pointed to by *`ptr`*. If *`ptr`* is the name of a statically allocated shared array, then the

[5] This discussion can readily be extended to regular C pointers, but this is out of the scope of this paper.

intended referent is obvious. Otherwise, minimal reaching-definition analysis is required to find where *`ptr`* was defined and therefore what the intended referent is. (We discuss later what happens if this analysis is not available, or if it fails.)

- If the intended referent is an array `t`, *`p`* must stay within the bounds of that array in the scope of the pragma. Before each expression dereferencing *`p`*, the compiler inserts code that checks inequalities (1) and either (2) or (3).

  When the second syntax is used, i.e., when X is specified, the compiler inserts additional code that checks

  $$p.\mathtt{addr} < \mathtt{t.addr} + \mathtt{X} \tag{5}$$

  before each dereference of *`p`*.

- If *`ptr`* is defined by a call to a memory allocation library procedure `upc_local_alloc`, `upc_global_alloc` or `upc_all_alloc`, then on entry to the scope, the compiler declares a temporary and saves the value of *`ptr`*:

  ```
  type * t = ptr;
  ```

  Before each dereference of *`p`*, the compiler inserts code that checks that *`p`* satisfies (4) or (5), depending on the allocation primitive, with respect to `t`. When X is specified, the compiler inserts, in addition, code that checks (5).

- If the analysis fails and the intended referent is not known, then only the second form (the one with `X)` can be enforced. As in the previous case, the compiler introduces code at the beginning of the scope that declares a temporary `t` where the value of *`ptr`* is saved, and enforces (5).

Notice also that both *`p`* and *`ptr`* must be declared outside the scope where the pragma applies. *`ptr`* must in addition be defined outside that scope. The scope doesn't have to include all references to *`p`*. If *`p`* is itself an element of an array of pointers, then all pointers in that array must have the same intended referent.

As an example, pointer `t2` in the running example is meant to point to `t`. The pragma could be inserted anywhere after line 6 and before line 22. Let's assume it's inserted just after line 14 and its scope goes up to an including line 18. The code then becomes:

```
14b   #pragma sticky t2:t
14c   {
15      t2 = &t[MYTHREAD];
16      t2++;
17      printf("Thread %d: t2=%p\n", MYTHREAD, t2);
18      *t2 = 0;
18b   }
```

The compiler introduces instructions that check inequalities (1) and (2) before each dereference of `t2` in the scope — in this example program, before line 18. These inequalities yield:

$$\mathtt{t.addr} \leq \mathtt{t2.addr} \leq \mathtt{t.addr} + 0 * \mathrm{sizeof}(\mathtt{double})$$

Actually, since `t2` is monotonically increasing on each thread, the compiler only needs to (insert code to) check the upper bound:

```
t2.addr ≤ t.addr
```

Thread 1 detects this inequality is not satisfied just before executing line 18, before `t2` corrupts array `p`.

The pragma extended with X is useful within called procedures, because this is typically a case where the intended referent is lost. Consider for instance the following procedure:

```
foo( type shared * t ) {
   type shared * p = t;
   #pragma sticky p:t:X {
      ..
```

In this procedure, if the compiler doesn't have information on the actual parameter(s) at the call site(s), it can still enforce that p never exceeds the value of formal parameter `t`, plus X. X may be passed as an additional argument to procedure `foo`.

Finally, notice that we do not check that the intended referent is still allocated.

6. LOCALIZED POINTERS-TO-SHARED

Let us come back to `t2` and assume, for the sake of the exposition, that the bounds checker misses that bug, or that there is no out-of-bound access. An issue in the code is still that another intent of pointer `t2` was lost: That intent is to refer to elements of shared array `t` that have affinity to thread `MYTHREAD`. Notice that "localizing" a pointer-to-shared does not defeat the purpose of pointers-to-shared. Typically, work on a shared array is divided up among threads such that each thread works on its local piece of the array, so that communication is reduced and performance improved. Another way to look at localized pointers is that they are pointers-to-shared that can be cast into a pointer-to-private. (Remember that if a pointer-to-shared is cast into a pointer-to-private on a thread that has no affinity with the shared object, the result is undefined.)

To make localization explicit, we introduce a new pragma, `localized`. Its syntax is:

```
#pragma localized ptr
{
   // Scope where the pragma applies
}
```

It requests the compiler to insert the following test before each dereference of pointer-to-shared *`ptr`*:

```
if( ptr.thread != MYTHREAD )
   error();
```

This pragma can be inserted anywhere after the definition of *`ptr`*. The scope does not have to include all references to *`ptr`*.

In the running example, a possible location for this pragma is again just after line 14, and for its scope to include lines 15 through 18:

```
14b   #pragma localized t2
```

```
14c   {
15       t2 = &t[MYTHREAD];
16       t2++;
17       printf("Thread %d: t2=%p\n", MYTHREAD, t2);
18       *t2 = 0;
18b   }
```

This information asks the compiler to insert code, before each dereference of `t2` within the scope, to check that `t2` points to local portions of any shared object (in contrast to a sticky pointer, that object need not stay the same). In this example, this code should be inserted immediately before line 18. Performing such a (run-time) check is usually inexpensive, since it boils down to checking that the thread field of the pointer equals `MYTHREAD`.

Moreover, static analyses can catch many cases. For instance, consider line 15 of the running example. `t2.thread` equals `MYTHREAD` mod `THREADS`, i.e., `MYTHREAD`. The compiler can thus easily check at compile-time that `t2` is local after line 15. Then, since `t` is allocated cyclically (its block factor is 1), `t2` cannot point to a local portion of `t` after the increment on line 16. `t2` is not modified before its dereference on line 18, so the compiler can at this point issue a warning or an error message.

This observation can be generalized as follows. Consider the following statement:

```
p += I;
```

where I is an expression, and assume `p` initially points to local data.

- If the block factor of the referent of `p` is 1, `p` keeps its locality property if and only if I is a multiple of `THREADS`. The compiler may or may not be able to assert this fact depending on the expression I and/or on whether the environment is static.
- If the block factor is greater than 1, then the compiler checks two cases. If the compiler can prove that I is such that

  $$B \leq I \leq B * (\texttt{THREADS} - 1)$$

  then `p` will lose its locality. On the other hand, if I is a multiple of B*`THREADS`, `p` remains local. Otherwise, the compiler cannot tell statically if locality is preserved.

Of course, the `localized` pragma can also apply to pointers whose intended referent is allocated dynamically. When the allocation is done by `upc_local_alloc()`, the entire allocated chunk of memory is local to the calling thread so the `localized` pragma probably doesn't help much. However, if the intended referent of pointer `p` is allocated by `upc_global_alloc(N,s)` or `upc_all_alloc(N,s)`, then we again have the property that if `p` points to local data, then `p` still does after

```
p += I;
```

if and only if `I` is a multiple of `THREADS`.

Note that, in the running example, either `localized` or `sticky` can help fix the issue with pointer `t2`. However, this is not always true. In fact, making a pointer both `localized` and `sticky` can be complementary. Consider the following code snippet:

```
  a = upc_all_alloc(N*THREADS, s);
  #pragma localized p
  #pragma sticky p:a
  { .. }
```

If `p` were ***only*** localized, the compiler could only infer that `p.thread` must equal `MYTHREAD` before each dereference of `p`. Symmetrically, if `p` were ***only*** sticky to `a`, let `a0` be the address of the first allocated element that sits on thread `MYTHREAD`[6]. (This assumes there is at least one element allocated on `MYTHREAD`, i.e., `N`>0). The compiler could only infer that:

$$\texttt{a0.addr} \le \texttt{p.addr} \le \texttt{a0.addr + (N-1)*s}$$

However, since `p` is both sticky and localized, we have a stronger property:

$$\texttt{a0} \le \texttt{p} \le \texttt{a0 + (N-1)*s}$$

assuming `thread`, `phase` and `addr` are bit fields of global pointers, in this order from highest to least significant. This result implies both earlier properties.

Finally, observe that `localized` suppresses generation of code to communicate to non-local storage.

7. REDUCING OVERHEAD

There is a well-known trade-off between the granularity of program safety and run-time overhead. We argue that checks may not need to be inserted immediately after each reference. In other words, we suggest a weakening of the semantics from "always perform bounds checks, and do them immediately" to "perform bounds checks opportunistically, possibly only when we can do so at no overhead", the degree of weakening being controlled e.g. by a command line switch. If the instruction scheduler (in the compile) keeps the right not to honor some of the checks (e.g., it may check only the upper bound on a pointer and not its lower bound), we show that the run-time overhead can be zero by construction — without of course eliminating all the checks. The safety mechanism is therefore weakened (because an error may not be caught, or may be caught too late), but on the other hand has a controlled overhead.

Clearly, the contract between the user and the compiler is then changed. More precisely, the compiler's instruction scheduler can be instructed by a simple command-line switch to enforce one of several policies, like enforcing all bounds and attributes, or on the contrary to enforce them only opportunistically.

Opportunistic scheduling of check instructions comes in at least two flavors: One leverages dead cycles, the other empty instruction slots.

[6] This address is implementation-dependent. It can however be computed by the following C statement:

```
  a0 = a + MYTHREAD;
```

Also, recall that `a0.addr = a.addr`.

Dead cycles are cycles where the processor's pipeline stalls, for instance when an instruction on the critical path is waiting for data. This phenomenon of course happens on both in-order and out-of-order microarchitectures, but the benefit, in our context, of in-order processors is that dead cycles are usually known to the compiler scheduler. Dead cycles can therefore be used to execute any sort of additional safety- or sanity-checking code in general, and, in particular, enforce bounds checking, shared variable protection, and `localized` and `sticky` pragmas — at no additional cost. As an example, consider a load of floating-point value on the critical path of some Itanium2 code. The important point here is that, on the Itanium2 processor, floating-point data can't be stored on the first level cache, so the latency of a floating-point load is at least 5 cycles. Since the load is on the critical path, the scheduler knows it has 4 dead cycles between the load and the first instruction that uses its result.

Empty instruction slots are missed opportunities to issue instructions to idle functional units at any given cycle. In other words, empty instruction slots appear when instruction-level parallelism is too low to use the full width of the processor. These empty slots give the scheduler the opportunity to squeeze in optional work at zero run-time overhead.

Consider the protection of `p[MYTHREAD]`. Before line 18b, both `p[MYTHREAD]` and `saved_p[0]` must be loaded. `p[MYTHREAD]` must be loaded anyway for the `printf` on line 19, so only the load of `saved_p[0]` is added. This load can be scheduled in the same cycle, at no additional cost:

```
    ld8 r39 = [r29]
    ld8 r40 = [r30] ;;
```

where `r29` and `r30` contain the addresses of `saved_p[0]` and `p[MYTHREAD]`, respectively. The values of these 8-byte elements (hence `ld8`) are stored in `r39` and `r40`, respectively. Then, instead of simply calling the library code for `printf`, like the original code would do:

```
     br.call  b0=printf# ;;
```

checking protection can be done as follows:

```
     cmp.neq  p4, p5 = r39, r40
(p4) br.call.spnt b1=error#
(p5) br.call.sptk b0=printf# ;;
```

The first instruction compares (`cmp`) `r39` and `r40`, and sets 1-bit predicate registers `p4` and `p5` to 1 and 0, respectively, if these registers are not equal (`neq`). If they are equal, `p4` is set to 0 and `p5` to 1. As a consequence, exactly one of the two predicate registers is true after the compare, and exactly one of the calls is executed: Either the call to the error procedure that is part of our protection mechanism, or the original `printf`. Notice that a single double-semicolon is needed, at the very end of the instruction sequence: All three instructions, including the set and use of predicate registers, can execute in exactly one cycle, just like the original non-predicated instruction calling `printf`. Notice also that there is no risk of branch misprediction thanks to branch hints suffixed to branch instructions: They tell the hardware the branch to `error` is statically predicted not taken (`spnt`), and that the branch to `printf` is statically predicted taken (`sptk`). The original and most probable control-flow path (through `printf`) is followed at full speed without mispredicts. If there is an error, the branch to `error` is mispredicted, but in this case performance doesn't matter any more.

Observe that, if the cost of executing partial checks can be reduced to zero, techniques described in this paper may still have an associated cost. For instance, the back-up storage required for protection may degrade caching behaviors and impact the TLB. Also, the checks mandated by `protect`, `localized` or `sticky` may increase register pressure.

8. IMPLEMENTATION & FUTURE WORK

Our partial and preliminary implementation is based on Berkeley's UPC compiler [13], itself based on the Open64 compiler [14]. (The test run shown in Section 2, however, was performed on an Alpha cluster and the code compiled with the HP UPC compiler.) The Berkeley UPC compiler is a source-to-source compiler that produces C code, which is compiled by a native compiler. Its authors found that the generated C code could be optimized satisfactorily by the native compiler, which justifies their choice not to hook up to Open64's WHIRL intermediate representation. In our case, however, and in order to validate the techniques of Sections 7 and 8 and measure their overhead, this hook up will need to be done.

9. RELATED WORK

Race detection is usually done in one of three ways: Statically, i.e. at compile-time; Dynamically, a.k.a on-the-fly [7][8][9][10][11]; and post-mortem a.k.a at replay time. Several techniques allow for both on-the-fly and post-mortem detections [25][26].

Work on static race detection includes [17][19][20][21]. Because the data races we address are due to erroneous pointer arithmetic, an instrumentation tool like that of [19], which identifies variable references that are never involved in a data race, would be defeated. For the same reason, an extended type system like that of [20] probably wouldn't fit our purpose. Because UPC is not object-oriented, [21] is not suitable either. As in [17], we rely on user annotations, but the compiler performs minimal analysis and generates code to detect races on-the-fly. Like e.g. [25] but unlike e.g. [10], our protection scheme only detect races that appear in a given execution, and never reports false positives.

One key difference between this paper and related work is that protection requires the user to specify the data structures that should be monitored. Future work might try to lift this restriction, but we are not convinced it would be realistic in the face of erroneous arithmetic on pointers-to-shared. Indeed, none of the papers we are aware of applies to multithreaded languages with arithmetic on pointers to shared objects.

ACKNOWLEGEMENTS

Brian Wibecan provided many very helpful comments on earlier drafts of this paper.

LEGAL NOTES

Patent applications have been filed on aspects of this work. Contact the authors for details.

REFERENCES

[1] T. El-Ghazawi, Th. L. Sterling, W. W. Carlson. UPC: Distributed Shared-Memory Programming. John Wiley & Sons. To appear, July 2003.

[2] W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks and K. Warren. Introduction to UPC and Language Specification, CCS-TR-99-157. IDA/CCS, Bowie, Maryland. May 1999.

[3] T. El-Ghazawi, W. W. Carlson and J. M. Draper. UPC Language Specifications V1.1 Second Printing, May 16, 2003.

[4] G. B. Bell, K. M. Lepak and M. H. Lipasti. Characterization of silent stores. In Intl Conf on Parallel Architectures and Compilation Techniques, Oct 2000.

[5] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. Software Practice and Experience, 1996.

[6] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. AADEBUG'97, Linköping, Sweden.

[7] E. Pozniansky and A. Schuster. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. In Proc. of Princ. & Practice of Parallel Prog, PPoPP'03, pp 179—190. , San Diego, California. June 2003.

[8] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In Proc. of Princ. & Practice of Parallel Prog, PPoPP'03, pp 167--178. , San Diego, California. June 2003.

[9] M. Rinard. Analysis of Multithreaded Programs. Lecture Notes in Computer Science, Vol. 2126. 2001.

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and Th. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In ACM Trans. On Comp. Sys. Vol 15. No. 4, Nov 997, pp 391—411.

[11] J.D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In Proc. of PLDI'02, June 2002, Berlin, Germany.

[12] A. Sălcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In Proc. of Principle & Practice of Parallel Prog, PPoPP'01. Snowbird, Utah. June 2001.

[13] W.-Y. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In Proc of Intl. Conf. on Supercomputing ICS'03. San Francisco. June 2003.

[14] Open64 Compiler Tools. http://open64.sourceforge.net

[15] D. Bonachea. GASNet Specification, v1.1. Tech Report CSD-02-1207. UC Berkeley. 2002

[16] S. S. Muchnick. Advanced Compiler Design & Implementation. Morgan Kaufmann, 1997.

[17] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata. Extended Static Checking for Java. In Proc. of Prog. Lang. Design & Impl. PLDI'02. Berlin, Germany. June 2002.

[18] GNU UPC. A GCC-based Compilation and Execution Environment for the Unified Parallel C Language. George Washington University. http://www.gwu.edu/~upc/software/gnu-upc.html

[19] J. Mellor-Crummey. Compile-time Support for Efficient Race Detection in Shared-Memory Parallel Programs. In In Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 129-139, San Diego, CA, May 1993.

[20] C. Flanagan and S. N. Freund. Type-based Raced Detection for Java. In Proc. of Prog. Lang. Design & Impl. PLDI'00, Vancouver, Canada. June 2000.

[21] Ch. von Praun and Th. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In Proc. of Prog. Lang. Design & Impl. PLDI'03, San Diego, California. June 2003.

[22] A. Goel, A. Roychoudhury and T. Mitra. Compactly representing parallel program executions. In Proc. of Principle & Practice of Parallel Prog, PPoPP'03, San Diego, California. June 2003.

[23] HP UPC Compiler. http://h30097.www3.hp.com/upc

[24] C. Cowan et al. Protecting Systems from Stack Smashing Attacks with StackGuard. In Proc. of the Linux Expo, Raleigh, NC, May 1999. http://www.immunix.org

[25] M. Ronsse, K. De Bosschere. RecPlay: a fully integrated practical record/replay system. ACM Transactions on Computer Systems (TOCS), Volume 17 Issue 2. May 1999.

[26] R. H. B. Netzer and S. Ghosh, Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization, *International Conference on Parallel Processing*, St. Charles, IL. August 1992.

[27] R. H. B. Netzer and B. P. Miller, On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions. *International Conference on Parallel Processing*, St. Charles, IL. August 1990.