October 14, 2003
RT0550
Computer Science 6 pages

Research Report

Array Bounds Check Elimination Utilizing a Page Protection Mechanism

Motohiro Kawahito

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Array Bounds Check Elimination Utilizing a Page Protection Mechanism

Motohiro Kawahito
IBM Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan
jl25131@jp.ibm.com

1. INTRODUCTION

In the past, there have been many research projects to eliminate array bounds checking. These approaches eliminated an array bounds check only if the array access corresponding to the check is known to be always valid. This paper presents a technique for eliminating an array bounds check even if it cannot be determined whether or not the corresponding array access for the target bounds check is always valid. We utilize a page protection mechanism to support this elimination. This technique can eliminate more array bounds checks than previous approaches.

Our approach first allocates both a normal memory area and a protected area at the time of array allocation. It next analyzes whether a memory operation always accesses the protected area when the memory operation performs an invalid access. If this condition is satisfied, we eliminate the array bound check for the memory operation.

This paper also covers how to limit the target arrays in order to reduce memory consumption and the cost of setting up this page protection. We select arrays by using two characteristics, the number of elements and life expectancy. Using these selection criteria, small and short-lived array objects are normally allocated.

2. PREVIOUS WORK

Previous approaches eliminated an array bounds check only if the array access corresponding to the check is known to be always valid. Eliminations using dataflow analysis [2, 3], eliminations using dependence analysis [1], and loop versioning [3] all fit in this category. However, these approaches still leave the array bound check of **data[i]** in Figure 1. This is because the exit condition of the loop depends only on the array's value.

```
int getLength(byte data[]) {
    int i;
    for (i = 0; data[i] != 0; i++); /* bound check still remains */
    return i;
}
```

Figure 1. Motivating example

3. OUR APPROACH

Our approach can eliminate the bound check in the example of Figure 1 by utilizing a page protection mechanism. Section 3.1 describes our basic idea for eliminating array bound checks utilizing the page protection mechanism. Section 3.2 describes how to limit the target arrays in order to reduce memory consumption and the cost of setting up page protection.

3.1 Basic Idea

Our basic idea consists of three steps:

1. We allocate both a normal memory area and a protected area at the time of array allocation.
2. We analyze whether a memory operation always accesses the protected area when the memory operation performs an invalid access. If this condition is satisfied, we eliminate the array bound check for the memory operation.
3. For Java, we perform exception transaction of ArrayIndexOutOfBoundsException in the handler of page protection fault.

Figure 2 shows a typical example of Step 1. We allocate an array at the page boundary in its tail and next allocate a protected page. If footer is required for the array, we allocate the footer after the protected page.

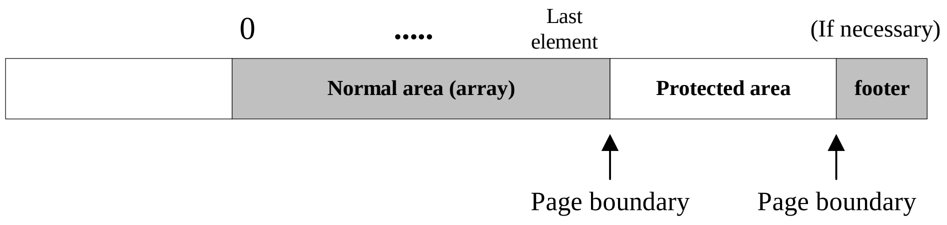

Figure 2. Example for Step 1

For Step 2, we target the frequently used loops with a single entry point and where the induction variable is changed at the end of the loop body. The example in Figure 1 is included in this category. We next collect range information for both minimum and maximum offsets from the induction variable for the array accesses whose subscript expressions are similar to “induction variable + offset” within the always executed path in the loop body. In this paper, the variables **min** and **max** denote these minimum and maximum offsets from the induction variable, respectively. If there is no such array in the target loop or if the target loop includes a try region, then we skip the loop for this optimization.

Next, we check the following two conditions for the target loop. In this paper, **ProtectN** is defined as (the size of the protected area / the size of each array element).

- –min <= initial value of the induction variable <= ProtectN – max – 1
- 1 <= incremental value of the induction variable <= ProtectN

If these conditions are satisfied, we can eliminate array bounds checks for “induction variable + offset” where the offset satisfies the following conditions.

min <= offset <= offset_max, where
offset_max = MIN(ProtectN - initial value of the induction variable – 1,
ProtectN - incremental value of the induction variable + max)

Finally, if a page protection fault is raised, we perform exception transaction of ArrayIndexOutOfBoundsException in the handler.

For the example in Figure 1, the variables used in this optimization are as follows:

min = max = 0;
ProtectN = 4096;
initial value of the induction variable = 0;
incremental value of the induction variable = 1;
offset_max = 4095;

Since offset_max is 4095, in this example we can eliminate array bound checks for data[i+0] through data[i+4095]. Therefore, we can eliminate the array bound check in the example in Figure 1.

<u>Another variation (countdown loops):</u>

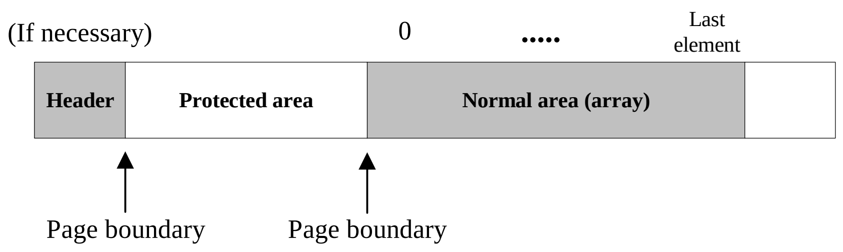

Figure 3. Memory allocation for countdown loop

If we allocate memory as shown in Figure 3, we can eliminate array bounds checks in countdown loops.

3.2 Limiting target arrays

If we perform the actions of the previous section for all arrays, the memory consumption and costs of setting up page protection may have a large impact. To avoid this, we need to select only effective arrays for this optimization. In this paper, we select such arrays by using two characteristics: (1) the number of elements and (2) life expectancy. This selection consists of four steps:

A) One kind of candidate for effective arrays would be arrays that have a large number of elements. This is based on the prediction that array bounds checking code may be frequently executed for arrays having many elements. Thus, if array allocation is requested with a large allocation size (e.g. 10,000 elements), we select the special allocation described in the previous section and set the flag denoting the special allocation in the array header.

B) We select effective object variables if the optimization in Section 3.1 is performed. This selection can be done by calculating scores for each variable. We use the following algorithm.

```
candidates = method parameters and global variables that can be moved to the entry of the method;
for (each V ∈ candidates){
   SCORE[V] = 0;    /* score of V */
   for (each U ∈ uses of V){ /* Use DU-chain */
      if (U is array bound check && definition of U is single){ /* Use UD-chain */
         if (V is special allocation, U can be eliminated){  /* analysis in the previous section*/
            SCORE[V] += execution frequency of U;      /* consider loop nesting and rare paths */
         }
      }
   }
}
if (the largest SCORE[V] <  threshold){
   We skip this method for this optimization. /* END */
} else {
    FirstV = V of the largest SCORE[V];        /* FirstV is used in the step C*/
}
```

C) For the candidate variable FirstV selected in the Step B), we check the flag described in Step A) at the method entry point. If an array object in the FirstV is specially allocated, this method will be recompiled with the assumption of special allocation for the FirstV. If this method has been recompiled, the compiled code will be simply invoked. For example, the following code will be inserted at the method entry.

```
if (the object in FirstV is specially allocated){           /* checking the flag in the header */
   Recompile and invoke the method with the assumption of special allocation for the FirstV.
} else {
   Invoke the normal compiled code.
}
```

D) The other candidates for effective arrays would be long-lived arrays. This is based on the prediction that array bounds checking code may be frequently executed for such array objects. In other words, we do not want to pay the cost of setting up page protection for short-lived array objects. In generational GC schemes, new objects are allocated to the young generation area and are advanced to the old generation area after surviving a small number of collections. This advancement is called *tenuring*. It is known that tenured objects tend to be long-lived. Therefore, we can move long-lived array objects to special allocation at the time of tenuring. This allows eliminating more array bounds checks.

By both Steps A) and D), small and short-lived array objects are allocated in the normal manner, and thus we can reduce memory consumption and the costs of setting up page protection.

As another idea, we can select arrays by the class of their elements. The following algorithm calculates scores for each class of element. SCORE will be added up for every compiled method. If SCORE[CLASS] is over a threshold, an array of CLASS will be allocated in the special manner.

```
for (each C ∈ array bounds checks in the method){
  CLASS = class of element for the array object of C
  if (array object of C is special allocation, C can be eliminated){  /* analysis in the previous section*/
    SCORE[CLASS] += execution frequency of C;        /* consider loop nesting and rare path */
  }
}
```

4. REFERENCES

[1] R. Bodik, R. Gupta, V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. *Conference on Programming Language Design and Implementation*, pp.321-333, 2000.

[2] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, Vol. 2, Nos. 1-4, pp.135-150, March-December 1993.

[3] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, T. Nakatani. Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal*, Vol. 39, No. 1, 2000.