Affine Parallelization of Loops with Run-Time Dependent Bounds from Binaries

Aparna Kotha, Kapil Anand, Timothy Creech, Khaled ElWazeer,
Matthew Smithson, and Rajeev Barua

University of Maryland,
College Park, MD 20742
{akotha,kapil,tcreech,wazeer,msmithso,barua}@umd.edu

Abstract. An automatic parallelizer is a tool that converts serial code to parallel code. This is an important tool because most hardware today is parallel and manually rewriting the vast repository of serial code is tedious and error prone. We build an automatic parallelizer for binary code, *i.e.* a tool which converts a serial binary to a parallel binary. It is important because: (i) most serial legacy code has no source code available; (ii) it is compatible with all compilers and languages.

In the past binary automatic parallelization techniques have been developed and researchers have presented results on small kernels from polybench. These techniques are a good start; however they are far from parallelizing larger codes from the SPEC2006 and OMP2001 benchmark suites which are representative of real world codes. The main limitation of past techniques is the assumption that loop bounds are statically known to calculate loop dependencies. However, in larger codes loop bounds are only known at run-time; hence loop dependencies calculated statically are overly conservative making binary parallelization ineffective.

In this paper we present a novel algorithm that enhancing past techniques significantly by guessing the most likely loop bounds using only the memory expressions present in that loop. It then inserts run-time checks to see if these guesses were indeed correct and if correct executes the parallel version of the loop, else the serial version executes. These techniques are applied to the large affine benchmarks in SPEC2006 and OMP2001 and unlike previous methods the speedups from binary are as good as from source. We also present results on the number of loops parallelized directly from a binary with and without this algorithm. Among the 8 affine benchmarks among these suites, the best existing binary parallelization method achieves an average speedup of 1.74X, whereas our method achieves a speedup of 3.38X. This is close to the speedup from source code of 3.15X.

Keywords: Automatic Parallelization, Binary Rewriting, Affine loop parallelization, Run-time dependent loop bounds.

1 Introduction

With the advent of multi-core machines it is most efficient to run parallel code on them. However, most code ever written is serial. Several methods have been

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 554–574, 2014.

proposed to parallelize serial code which include: (i) explicitly rewriting serial code using Message Passing Interface (MPI), pthreads, Threading Building Blocks (TBB) etc; (ii) using program directives such as Open Multi-Processing (OMP) to specify parallelism in serial code and (iii) using an automatic parallelizer to convert serial code to parallel without any human intervention. Automatic parallelizer is more attractive than the first two methods since: (i) it is not prone to human error; (ii) programmers do not need to be trained to think and program in parallel. Hence, we choose automatic parallelization to bridge the gap between serial code and parallel hardware.

In this paper we develop mechanisms to implement an automatic parallelizer within a binary rewriter. i.e. we develop *a tool that takes as input serial binary code and produces as output a parallel binary.* The advantages of parallelizing binary code include: (i) it works on old legacy code for which no source code is available; (ii) it works for all binaries of an instruction set irrespective of the language/compiler they come from; (iii) it works on hand-coded assembly language programs as well; (iv) it can be used by the end user who does not have access to the source code.

In the past a few attempts have been made to parallelize affine codes directly from binaries Kotha et al. (2010); Pradelle et al. (2012). Though these papers present good foundational ideas and results on polybench kernels, they are only a start to parallelizing large real world affine codes. The major limitation of these methods is that their algorithms are not powerful enough to handle loops with run-time dependent loop bounds and such loops are present in abundance in real life code. In this paper we present a novel algorithm to work on such loops whose bounds are run-time dependent or statically unknown. The idea is that we guess the most likely loop bounds using the memory expressions present in the loop and add run-time checks to see if these were indeed correct before executing the parallel version of the loop. These run-time checks may slow down the program in the worst case but open up for more possible parallelism. Our results show that affine benchmarks from the SPEC2006 and OMP2006 benchmark suites (much larger benchmarks representative of real world codes) can be parallelized with our techniques.

Further, this paper is arranged as follows. Section 2 presents the closest related work contrasting our techniques to them. Section 3 presents the limitations of the present binary affine parallelization techniques using an example and motivates the algorithm followed by a brief algorithm and more examples in section 4. The core algorithm is presented in section 5 followed by the description of our infrastructure in section 6 and the results in section 7.

2 Related Work

In this section, we present potentially competing related work contrasting it to this paper in the following categories: (i) static automatic parallelization of binaries; (ii) dynamic automatic parallelization of binaries; (iii) automatic vectorization of binaries and (iv) array delinearization techniques.

Static Automatic Parallelization of Binaries: Kotha et al. (2010) and Pradelle et al. (2012) are the only two static methods we are aware of that have done automatic parallelization in a binary rewriter. Both these methods present results on small kernels that are a part of the polybench benchmark suite. Pradelle et al. (2012) automatically parallelizes binaries by feeding the binary intermediate form to the polyhedral compiler. Its results are only on the polybench benchmark suite. Kotha et al. (2010) statically parallelizes binaries by using dependence information determined from binaries. However, its methods are limited to affine loops where loop bounds are known and hence, it also can only parallelize small kernels from the polybench benchmark suite. Both these methods present a brief section on run-time dependent loop bounds and suggest adding run-time checks to check if different accesses were indeed to different arrays. Their methods are highly lacking since they have no mechanisms to reason about dependencies between accesses to the same array and in the absence of such a mechanism it will be conservative and not parallelize real world code. We build on this work and have devised a novel algorithm to guess the possible loop bounds of affine loops. We are able to parallelize affine benchmarks from the SPEC2006 and OMP2001 benchmark suites.

Dynamic Automatic Parallelization of Binaries: Dynamic automatic parallelization techniques present in literature are Yardimci and Franz (2006), Wang et al. (2009) and J. Yang and Whitehouse (2011). Yardimci and Franz (2006) focuses on a dynamic method to detect non-affine parallelism. Wang et al. (2009) presents a dynamic method to parallelize binaries using speculative slicing and J. Yang and Whitehouse (2011) presents a method to use run-time information to parallelize binary code. All the three methods are dynamic. Hence, they suffer from run-time overheads from analysis. Most importantly, they do not optimize for affine loops whereas our method does.

Vectorization of Binaries: Nakamura et al. (2011) and Dasgupta and Dasgupta (2003) present techniques to analyze binaries and vectorize them. Their analysis is limited to vectorization of binaries and do not attempt to parallelize using threads like we do.

Array Delinearization Techniques: Array Delinearization methods Maslov (1992) Franke and O'boyle (2003) take source code with linearized multi-dimensional accesses, and convert them to multi-dimensional accesses when possible. Ideally, if we delinearize array accesses in a binary we can parallelize them as effectively as from source. However, source-level methods to delinearize array accesses cannot be adapted to binaries easily. Delinearization methods such as Maslov (1992), Franke and O'boyle (2003) require high level intermediate C like representation which is not available from binary code. They use symbolic information which contains information about the number, location, and dimension sizes of arrays, to delinearize arrays. Finding this information in the general case from stripped binaries (*i.e.* those without symbolic information) is impossible since it is discarded by the linker. Hence, delinearization methods cannot be adapted for binary code.

The method in this paper circumvents the problem of missing array information in binaries by not attempting to recover guaranteed information about array locations and dimension sizes. Instead it guesses the possible bounds for loops. When the guesses are correct, the code can be parallelized. Run-time checks ensure that when the guessed bounds are wrong, the serial code is executed. No previous method guesses loop bounds from binaries, or uses run-time checks like our method. *The result is that our method is the first to parallelize binary code with unknown loop bounds.*

3 Motivation

To parallelize affine loops, traditional techniques calculate distance vectors for each loop and use them to reason about parallelizing the loop. In this section we first describe the best-known methods for obtaining distance vectors from source code for affine loops with run-time determined loop bounds. We then present the limitations of the existing binary method for the same and briefly describe our method.

```
int A[20,50]
for i = 0 → ub_i step 1
  for j = 0 → ub_j step 1
    A[i,j] = A[i,j] + 10
```

The code shows a normalized loop, *i.e.* a loop with a lower bound of zero and a step of one. Loops can be normalized using existing methods such as the normalization pass in LLVM.

Fig. 1. Code Example

Distance vectors from source for code in figure 1 are calculated as follows. Existing methods make the assumption that row and column accesses are within the bounds of the array's dimensions. They solve for two iterations that refer to the same memory location in the infinite space for each dimension separately. If no solution exists, like in this example, they conclude that no two iterations ever access the same memory. This implies that iterations of the j loop can execute in parallel (i.e., the component of the distance vector for j dimension is zero.) Similarly, they proves that the loop i is parallel.

To obtain distance vectors from binary for this code we cannot use the above source method since it relies on known affine expressions for array indexes in terms of induction variables, which are not apparent from the binary. Instead we start with the existing method for binaries in Kotha et al. (2010). It shows that we can recover linearized expressions for memory accesses from a binary, and solving these linearized expressions gives us distance vectors. In the presence of loop bounds the solutions from binaries are very powerful, and can handle most linear algebraic kernels as presented in Kotha et al. (2010). However, when loop bounds are run-time dependent, we need to solve these linearized expressions in the infinite space (since we need to assume that the loop bounds can take any value at run-time). This greatly reduces the precision of the analysis.

Let us apply the existing binary method to code in figure 1. From its binary, we recover a memory expression of the form $\texttt{Base}_\texttt{A} + \texttt{200i} + \texttt{4j}$ which corresponds to the A[i,j] access (assuming the element size is 4). The "200" in $200i$ is because

the size of a row is 50 elements, each of 4 bytes. We need to reason about this access in the infinite space for `i` and `j` since the loop bounds are unknown. In the infinite space, iterations (2, 0), (1, 50) and (0, 100) refer to the same memory location. All the iterations except (2, 0) are not possible since the legal range of j is [0,49] and beyond 49 the code accesses columns out of bounds wrapping into rows. Source code methods assume that such iterations are not possible; hence proving the loop is parallel. However, the binary method in Kotha et al. (2010) cannot make any such assumptions about iterations remaining within array bounds, since array bounds and dimensions are not known. As a result, without loop bounds, the binary method in Kotha et al. (2010) fails to prove that this loop is parallel because false loop-carried dependence appears.

In this paper we present a method to statically guess the most likely upper bounds of loops when loop bounds are statically unknown. We then use run-time checks to see if the loop bounds were indeed within the guessed range and execute the parallel version when the run-time checks succeed, else we execute the serial version. For example, for code in figure 1, using the theory presented in Kotha et al. (2010) we discover the memory expression for the A[i][j] access to be $\texttt{Base}_\texttt{A} + \texttt{200i} + \texttt{4j}$. We then look at the coefficients multiplying the induction variables in this memory expression and guess that the likely limit of the induction variable with the smallest coefficient (*i.e.* `j`, as the immediately higher coefficient divided by the coefficient of this induction variable; *i.e.* in this example we guess the limit on `j` as (Coefficient of `i`/Coefficient of `j`) (*i.e.* $\frac{200}{4} = 50$). By guessing that `j` is less than `50` no two iterations will access the same memory location because now `j` has been prevented to wrap into `i`. At run-time, we check if `j` is indeed less than `50`. In this case, this check will always succeed and we will always execute the parallel version of the loop.

4 Examples

In this section we first briefly describe the steps of the algorithm described in section 5 and then apply it to four code examples to show how their loops can be parallelized from a binary even though the loop bounds are run-time dependent.

First, we state the algorithm that we use to guess the loop bounds for a loop directly from a binary and then present details in section 5.

Step 1: Divide memory accesses (both reads and writes) in a loop into *Dependence Groups (DGs)*. Intuitively, a DG is a subset of memory addresses in the loop that are sufficiently close to one another.

Step 2: Arrange all DGs in ascending order of their base addresses, from $\texttt{DG}_1$ to $\texttt{DG}_\texttt{T}$.

Step 3: For all the DGs that have writes in them make best guesses for the possible range for induction variables. These guesses are called intra-group constraints, since they are obtained by working on one DG at a time.

Step 4: Initiate worklist with DGs that have constraints remaining after step 3.

Step 5: Work on each $\texttt{DG}_\texttt{i}$ in the worklist and solve for the values of induction variables such that the accesses in $\texttt{DG}_\texttt{i}$ do not overlap with those in $\texttt{DG}_{(\texttt{i+1})}$.

This generates further guesses on the induction variables. Merge these new constraints with existing constraints for the same induction variable by choosing the minimum. These guesses are called inter-group constraints because they are obtained by constraining DG_i to not overlap $DG_{(i+1)}$.

```
int A[20,50]
int B[20,50]
for i = 0 → ubi step 1
  for j = 0 → ubj step 1
   B[i,j] = A[i,j] + 10
```

(a) Example 1

```
int A[20,50]
for i = 0 → ubi step 1
  for j = 0 → ubj step 1
    A[2i,j] = 10*i+j
```

(b) Example 2

```
int A[100]
for i = 0 → ubi step 1
  A[i] = i;
  A[i+50] = i + 50;
```

(a) Example 3

Example 1: The memory address expressions that we recover from the binary of example 1 are of the form $Base_A + 200i + 4j$ and $Base_B + 200i + 4j$ (Assuming that the size of an integer is 4.). $Base_A$ and $Base_B$ will at least differ by 4000, since the size of each array is 4000 bytes. Without loss of generality let's assume we recover the following from the binary $100 + 200i + 4j$ and $4100 + 200i + 4j$.

When the code above is compiled to a striped binary, all symbolic information is lost. Hence we no longer know the location or dimension sizes (20, 50) of array `A`. Hence we can no longer infer (as we implicitly do from source) that $\mathtt{ub_i} < 20$ and $\mathtt{ub_j} < 50$. Instead we must assume that the loop bounds can take any value.

We now show briefly how our algorithm is applied to these accesses to guess the bounds on i and j. In Step 1, we check to see if the accesses belong to different DGs. The heuristic we use is that the difference of the bases is greater than a factor (5 for our experiments) of the highest coefficient; *i.e.* $Base_B - Base_A > 5 \times 200$ *i.e.* (4100-100) $> 5 \times 200$. Since this is true both the accesses will belong to different DGs. In Step 2, we arrange the DGs in ascending order of their bases. $100+200i+4j$ belongs to DG_1 because its base is lower than the second access which belongs to DG_2. In Step 3, we solve for intra-group constraints in DG_2 since it contains a write. We guess the bound on j by dividing the co-efficient multiplying i (the just higher co-efficient in the linearized equation) by the co-efficient of j *i.e.* ($\frac{200}{4} = 50$). Hence, we guess that j must belong to [0, 49]. In step 4, we create a worklist with all DGs that have constraints remaining. In this example both the DGs have constraints remaining on i; hence both of them will belong to the worklist. In step 5, we guess the bound on i by solving that DG_1 *i.e.* $100+200i+4j$ does not overlap with DG_2 *i.e.* $4100+200i+4j$ given the highest possible value for j is 49; *i.e.* $100+200i+4*49 < 4100$. Hence, i must be less than 19.02 or in the range [0, 19]. Since DG_2 is the highest DG we do not solve for it overlapping with any other DG.

After we have applied our algorithm to this loop, our guess for i is [0, 19] and j is [0, 49]. We now solve for dependencies within this range for the loop and discover that the loop can be parallelized. We also add lightweight run-time checks before the parallel version of the loop (which will always succeed for this loop).

Example 2: The memory address expression that we recover from the binary in example 2 is $Base_A + 400i + 4j$. Since there is only one access, step 1 and 2 will result in placing it in DG_1. In step 3, we guess that the bound of j is ($\frac{400}{4} = 100$) or the range of j is guessed to be [0, 99]. There would be no step 4 and 5 for this loop since there is only one DG.

Next we calculate dependencies assuming the range of j is [0, 99] and i can take any value and discover that the loop can be parallelized. In reality however the range of j will not exceed [0, 49]. But our larger discovered bounds work well since even if they did exceed 49 and be below 99 this loop can still be parallelized *i.e.* if the programmer decided to access two rows using a column increment (which most programmers would not do) it is still a parallel loop. From the binary this means that we see array A of size [20,50] as an array of size [10,100]. However, this is fine since we reason about the dependencies in the correct way and parallelize the loop only when our run-time checks succeed.

Example 3: The equations we will recover from the binary of example 3 are $Base_A + 4i$ and $Base_A + 200 + 4i$. After step 1, we will place them in different DGs since the difference between the bases (200) is greater than 5 times the highest co-efficient 4. After arranging the DGs in ascending order in step 2, $Base_A + 4i$ will belong to DG_1 and $Base_A + 200 + 4i$ will belong to DG_2. No intra-group guesses are calculated in step 3 since the recovered equations are single dimensional. After step 4, the worklist is populated with both the DGs since both contain i for which there is no guess as yet. In step 5, we solve for inter-group guesses such that DG_1 does not overlap with DG_2, *i.e.* $4i < 200$ or $i < 50$. Hence, the range we guess for i is [0, 49] which is also the actual limit on i from source. The run-time check will always succeed in binary code and we will execute the parallel version of this loop. This is correct because, regardless of the value of ub_i, the two array references access non-intersecting portions of the array. Our method correctly treats these non-intersecting portions as different arrays.

5 Algorithm to Guess Loop Bounds

In this section we describe in detail the algorithm briefly presented in section 4. First we describe which loops from binary code we work on and then in subsequent subsections we describe the steps of the algorithm in detail.

First, we would like to present to you the kind of loops on which our algorithm is applied on and the kind of loops on which our algorithm is effective. We apply our method to every loop that has only affine accesses in them *i.e.* accesses of the form A[i+3][5j], A[i+j][k+i], A[j][j] etc are all processed by our method. We also apply our method only on loops whose bounds are loop invariant. Our method is able to effectively parallelize loop nests with array accesses of the form A[i][2j], A[3j][i+100], and A[j][i] *i.e.* normalized accesses with induction variables in any order; however affine accesses having multiple induction variables in a single array index expression (such as A[i+j]) or having repeated induction variables (such as A[j][j]) are not currently effectively parallelized by our method. Our guesses may be incorrect for these loops. Hence, the run-time checks might fail for these loops and the serial version of the code may be executed. However, these kinds of accesses are rare in real code and hence our method is nearly as powerful from binary as from source.

Every affine memory address that we recover from the binary is a linearized multidimensional equation of the form Kotha et al. (2010):

$$\texttt{MemAddr(Base, d)} = \texttt{Base} + \sum_{\texttt{j}=1}^{\texttt{n}} \texttt{d}_\texttt{j} \times \texttt{i}_\texttt{j} \tag{1}$$

(where Base and d's are constants or loop invariant quantities, i's are induction variables, and $\texttt{d}_1 >= \texttt{d}_2 >= \ >= \texttt{d}_\texttt{n}$). We arrange the memory expression with d's in this order since in the algorithm we use the immediately higher coefficient while guessing the value of a particular induction variable, *i.e.* we use $\texttt{d}_{(\texttt{m}+1)}$ when guessing the values of induction variable $\texttt{i}_\texttt{m}$. We will refer to memory addresses from binary using MemAddr(Base, d) throughout the paper. Different memory addresses from binary will have different Base and ds. Since we work on loops with only affine accesses in them, if we discover that a loop contains an access that is not affine *i.e.* we cannot discover a linearized expression for it then we do not work that loop.

In the following subsections we first describe our algorithm and then present an intuition for it.

5.1 Step 1: Divide the Accesses into DGs

A DG is a subset of memory references in the loop that are sufficiently close to one another and these set of references most likely do not overlap with other DGs. Intuitively, while dividing memory references into DGs we try to guess all the references which access the same array, or a region of an array not overlapping with other regions. This is not immediately apparent since binaries lack symbolic information containing the locations and sizes of arrays.

We create DGs using the following method. We look at the address of each memory reference and place it in an already present DG if it is sufficiently close to the addresses already in that DG; else we create a new DG with this memory address. We define that two accesses are sufficiently close to one another if the difference between the bases is within a factor (5) of the highest coefficient in the memory expression. The formal algorithm is presented in algorithm 1. We use a factor 5 which we find effective in most cases; however any other method can be used as well to determine which accesses are close to each other.

We now describe some of the terms used in the algorithm. DGlist is the list of DGs that is initialized to NULL and then populated as we consider every memory access in the loop. $\texttt{d}_1$ is the highest coefficient in the memory expression; hence, if the difference between the base and any of the bases already in a DG is within a factor of it, we guess that it most likely belongs to the same memory array and place this reference in that DG. $\texttt{CD}_{\texttt{Thres}}$ is a number that guesses the maximum difference between references in the same DG. Currently we set $\texttt{CD}_{\texttt{Thres}}$ to 5. With $\texttt{CD}_{\texttt{Thres}} = 5$, two accesses to A[i] and A[i+4] will belong to the same DG, whereas two accesses to A[i] and A[i+10] will belong to different DGs.

We manually looked at many affine benchmarks and determined that having accesses A[i] and A[i+e] where $e > 5$ in the same loop is relatively rare in affine codes; as most constants in affine codes are less than 5. Most of the codes only look at neighbouring values (*i.e.* use constants $\pm$ 2) to update an array. Hence, even if we had accesses to A[i+2] and A[i-2], the difference 4 is still lower than the factor 5 we

choose. If the rare case occurs, and there are accesses to A[i] and A[i+e] (e > 5), we would treat them as accesses to two different arrays. Accesses to different arrays A and B will belong to different DGs unless the highest dimension of A has size less than 5 (which again is very rare) and B immediately follows A in the binary's data layout. Most often in affine codes, the array sizes are relatively huge running into thousands. If this rare case appears we will treat both A and B as the same array. In both the above cases, the run-time checks will fail and the serial version of the loop will be executed. Hence, the loop may run slower than from source, but correctness is always maintained.

Algorithm 1. Step 1: Algorithm to divide accesses into DGs

```
  Input: MemAddr(Base, d) for all accesses in loop
  Output: DGlist has the accesses divided into DGs
Require: Initialize DGlist to NULL
  for all MemAddr(Base, d) in loop do
      Initialize TmpDGlist to NULL
    for all DG_i in DGlist do
      if |Base − Any base in DG_i|<d_1 × CD_Thres then
        Put MemAddr(Base, d) in DG_i
        Put DG_i in TmpDGlist
      end if
    end for
    if sizeof TmpDGList > 1 then
      Merge all the DGs in TmpDGList
    end if
    if sizeof TmpDGList == 0 then
      A new DG with MemAddr in it is added to DGlist
    end if
  end for
```

5.2 Step 2: Arrange DGs in Ascending Order

In this step we reorganize the DGs in `DGlist` in ascending order of the bases present in them. After arranging those in ascending order the following will be true:

All bases in $\texttt{DG}_1$ < All bases in $\texttt{DG}_2 < \cdots <$ All bases in $\texttt{DG}_\texttt{T}$ (This will be < since if they are equal they would belong to the same DG). We call this ordering of DGs, the *FullList*.

5.3 Step 3: Induce Intra-group Dependencies

In this step we make our best guesses for all array bounds, and hence induction variables, except the array bound of the highest dimension in an array reference. We make the guesses based on the assumption that array references accesses arrays within the bounds of each dimension.

We apply step 3 to every DG that has a write in it. The reason we apply it to DGs with writes in them is that even if a read accesses across bounds it does not create a

Algorithm 2. Step 3.1: Guesses for induction variables using one access

Input: All DGs that have a write in them
Output: Initial guesses for the induction variables
Require: Initialize each of $g_1, g_2, \cdots, g_n$ to TOP
for all DG_i in FullList that has a write in it **do**
  for all $MemAddr(Base, d)$ in DG_i **do**
    for $k = 2 \rightarrow n$ **do**
      $g_{1k} = \lfloor \frac{d_{(k-1)}}{d_k} \rfloor$
      $g_k = min(g_k, g_{1k})$
    end for
  end for
end for

loop dependency that prevents parallelization; hence guessing bounds considering DGs with only reads is not necessary. For example, if there is an affine loop that only reads from an array, there is no need to guess bounds for such a loop as it is parallel in the infinite space as long as there is no scalar dependency in it.

Step 3 is divided into two sub steps 3.1 and 3.2. Step 3.1 is applied to every access in a DG and step 3.2 is applied to a pair of accesses in a DG. We first present the algorithms for both the sub steps before presenting intuitions for them.

Step 3.1: The formal algorithm for step 3.1 is presented in algorithm 2. We are working on loop nests with induction variables say $\mathtt{i_1}, \mathtt{i_2}, \cdots, \mathtt{i_n}$ and guesses for each $\mathtt{g_1}, \mathtt{g_2}, \cdots, \mathtt{g_n}$. First, we initialize the guesses for each of these induction variables to `TOP` representing infinity which is what we know about each of the induction variables before the start of this step. Then we look at every memory access which is of the form `MemAddr(Base,d)` (from eq(1)) and make guesses for each induction variable as follows.

$$\text{The guess on } i_k,\ \mathtt{g_{1k}} = \lfloor \frac{\mathtt{d_{(k-1)}}}{\mathtt{d_k}} \rfloor\ \forall \mathtt{k} \in [2, \mathtt{n}] \tag{2}$$

We then update the guess already in $\mathtt{g_k}$ for $\mathtt{i_k}$ using

$$\mathtt{g_k} = \mathtt{min}(\mathtt{g_k}, \mathtt{g_{1k}}) \tag{3}$$

Note: $\mathtt{min}(\mathtt{TOP}, \mathtt{g_{1k}}) = \mathtt{g_{1k}}$ since `TOP` represents infinity.

We apply this to every memory access in every DG that has a write and guess for every induction variable other than the highest dimension $\mathtt{i_1}$. Note that we cannot make a guess for $\mathtt{i_1}$ since there is no $\mathtt{d_0}$ in the equation. Hence, we do not have a guess for $\mathtt{i_1}$ in this step. The guess for $\mathtt{i_1}$ is made in step 5 and will be described later.

Step 3.2: After we have applied step 3.1 to all DGs that have a write in them, we work on the same DGs considering pairs of accesses in them and apply step 3.2 on them. This algorithm is presented in algorithm 3.

We now describe the algorithm briefly. We first initialize $\mathtt{x_1}, \mathtt{x_2}, \cdots, \mathtt{x_n}$ to zeroes. These represent the adjustment we need to make to each of the induction variable bound guesses at the end of this step. Then we consider pairs of accesses in this DG, if the bases are different then we store the absolute difference in $\mathtt{Base_{diff}}$. We then run a loop that checks to see which factor of this difference came from which

Algorithm 3. Step 3.2: Guesses for induction variables using pair of accesses

Input: All DG_is that have a write in them and $g_1, g_2, \cdots, g_n$ from step 3.1
Output: Refined guesses for induction variables
Require: Initialize $x_1, x_2, \cdots, x_n$ to zeroes
for $MemAddr_1(Base_1, d)$,$MemAddr_2(Base_2, e)$ in DG_i **do**
  $Base_{diff} = \mid Base_1 \text{ - } Base_2 \mid$
  for $k = 1 \rightarrow n$ **do**
    if $\frac{Base_{diff}}{gcd(d_k, e_k)} \geq 1$ **then**
      $x_{11} = \lfloor \frac{Base_{diff}}{gcd(d_k, e_k)} \rfloor$
      $x_k = max(x_k, x_{11})$
      $Base_{diff} = Base_{diff} - \lfloor \frac{Base_{diff}}{gcd(d_k, e_k)} \rfloor \times gcd(d_k, e_k)$
    end if
  end for
end for
for $k = 1 \rightarrow n$ **do**
  $g_k = g_k - x_k$
end for

co-efficient and keep track of that in $\mathtt{d_k}$s. Later these are subtracted from the guesses for induction variables $\mathtt{g_k}$ from step 3.1.

It is important to make this adjustment to the guesses on loop bounds from step 3.1 since by doing so we are making sure that each of the accesses do not run into the higher dimension of the other. After this adjustment we will not have spurious dependencies from binary that prevent prallelization. We will present further intuition to this step below.

Intuition for Step 3.1: Let us assume that the binary code we are accessing came from source code where the loop nest had induction variables (say $\mathtt{i_1}, \mathtt{i_2}, \cdots, \mathtt{i_n}$) and an array accesses $\mathtt{A}[\mathtt{C_1} \times \mathtt{i_1} + \mathtt{B_1}][\mathtt{C_2} \times \mathtt{i_2} + \mathtt{B_2}] \cdots [\mathtt{C_n} \times \mathtt{i_n} + \mathtt{B_n}]$ in the loop and the size of array A is $[\mathtt{n1}][\mathtt{n2}] \cdots [\mathtt{nn}]$. Assume that none of the induction variables is repeated; however any ordering of the induction variables is allowed. This access when recovered from the binary will be of the form.

$$(\mathtt{Base_A} + \sum_{\mathtt{j}=1}^{\mathtt{n}} \mathtt{B_j} \times \prod_{\mathtt{m}=\mathtt{j}+1}^{\mathtt{n}} \mathtt{n_m}) + \sum_{\mathtt{j}=1}^{\mathtt{n}} \mathtt{C_j} \times \prod_{\mathtt{m}=\mathtt{j}+1}^{\mathtt{n}} \mathtt{n_m} \times \mathtt{i_j} \tag{4}$$

(This assumes an element size of 1; else each one of the terms will be multiplied by the element size.) The algorithm is correct even if the compiler uses the column-major layout; we assume the row-major layout only for explaining the intuition. Our results also include FORTRAN benchmarks for which the gfortran compiler uses the column-major layout. $\mathtt{Base_A}$ and all the terms containing $\mathtt{B}$'s (shown in parenthesis above) are rolled into the constant term when recovered from the binary. We know that the memory address that we recover from binary is of the form $\mathtt{MemAddr(Base, d)}$ (from eq.(1)).

Equating (4) and (1) we get:

$$\mathtt{Base} = \mathtt{Base_A} + \sum_{\mathtt{j}=1}^{\mathtt{n}} \mathtt{B_j} \times \prod_{\mathtt{m}=\mathtt{j}+1}^{\mathtt{n}} \mathtt{n_m} \tag{5}$$

$$\text{and, } \mathtt{d_j} = \mathtt{C_j} \times \prod_{\mathtt{m}=\mathtt{j}+1}^{\mathtt{n}} \mathtt{n_m} \tag{6}$$

First, let us calculate the actual upper bounds of the induction variables from source. From source we know that the array indices do not access arrays out of their bounds. Hence, each dimension index must be less than the actual size of that dimension.

$$i.e.\ \mathtt{C_k} \times \mathtt{i_k} + \mathtt{B_k} < \mathtt{n_k} \tag{7}$$

$$\text{Rearranging the terms, } \mathtt{i_k} < \frac{(\mathtt{n_k} - \mathtt{B_k})}{\mathtt{C_k}} \tag{8}$$

Hence, the upper bound of $\mathtt{i_k}$ from source is $\frac{(\mathtt{n_k} - \mathtt{B_k})}{\mathtt{C_k}}$.

Second, let us see what our guess for induction variable $\mathtt{i_k}$ is by applying step 3.1 to this access. Our guess for induction variable $\mathtt{i_k}$ is obtained by substituting eq(6) in eq(2)

$$i.e.\ \mathtt{g_{1k}} = \lfloor \frac{\mathtt{C_{(k-1)}} \times \mathtt{n_k}}{\mathtt{C_k}} \rfloor \ \forall \mathtt{k} \in [2, \mathtt{n}] \tag{9}$$

Next taking the minimum of $\mathtt{g_{1k}}$ and $\mathtt{TOP}$ (the initialized value) we get,

$$\mathtt{g_k} = \mathtt{min}(\mathtt{TOP}, \mathtt{g_{1k}}) = \mathtt{g_{1k}} = \lfloor \frac{\mathtt{C_{(k-1)}} \times \mathtt{n_k}}{\mathtt{C_k}} \rfloor \ \forall \mathtt{k} \in [2, \mathtt{n}] \tag{10}$$

We now show that the guesses for induction bounds are greater than or equal to the actual loop bounds. This is important because if the guesses were lower than the actual bounds our run-time checks would fail. We have already seen that the guess on the induction variable $\mathtt{i_k} = \frac{\mathtt{C_{(k-1)}} \times \mathtt{n_k}}{\mathtt{C_k}}$, this is greater than the actual limit of $\mathtt{i_k}$, which is $\frac{(\mathtt{n_k} - \mathtt{B_k})}{\mathtt{C_k}}$ from eq.(8). We observe that if $\mathtt{C_{(k-1)}}$ is 1 and $\mathtt{B_k}$ is 0, then the value we would have guessed is the same as the actual upper bound. Further, if $\mathtt{C_k}$ is 1 as well, the guess for $\mathtt{i_k}$ is $\mathtt{n_k}$, which is the size of that array dimension. Every guess we make for the induction variables is actually higher than or equal to its actual bound as shown above. By taking the minimum at every step we have a guess that is at least its actual bound.

Intuition for step 3.2: Let us assume that there is a second accesses to A, $\mathtt{A}[\mathtt{C_1} \times \mathtt{i_1} + \mathtt{B_1} + \mathtt{E_1}] \cdots [\mathtt{C_n} \times \mathtt{i_n} + \mathtt{B_n} + \mathtt{E_n}]$ in this loop where Es are small numbers < 5. The memory address for this access from binary will be of the form:

$$\mathtt{Base_A} + \sum_{\mathtt{j}=1}^{\mathtt{n}} (\mathtt{B_j} + \mathtt{E_j}) \times \prod_{\mathtt{x}=\mathtt{j}+1}^{\mathtt{n}} \mathtt{n_x} + \sum_{\mathtt{j}=1}^{\mathtt{n}} \mathtt{C_j} \times \prod_{\mathtt{x}=\mathtt{j}+1}^{\mathtt{n}} \mathtt{n_x} \times \mathtt{i_j} \tag{11}$$

Recollect that this access when recovered from the binary will be of the form $\mathtt{MemAddr}(\mathtt{Base_2}, \mathtt{e})$, from equation (1):

$$\mathtt{MemAddr}(\mathtt{Base_2}, \mathtt{e}) = \mathtt{Base_2} + \sum_{\mathtt{j}=1}^{\mathtt{n}} \mathtt{e_j} \times \mathtt{i_j} \tag{12}$$

Equating eq.(11) and eq.(12), we get:

$$\mathtt{Base_2} = \mathtt{Base_A} + \sum_{\mathtt{j}=1}^{\mathtt{n}} (\mathtt{B_j} + \mathtt{E_j}) \times \prod_{\mathtt{x}=\mathtt{j}+1}^{\mathtt{n}} \mathtt{n_x} \tag{13}$$

$$\text{and, } \mathtt{e_j} = \mathtt{C_j} \times \prod_{\mathtt{x}=\mathtt{j}+1}^{\mathtt{n}} \mathtt{n_x} \tag{14}$$

First, let us prove that both the references belong to the same DG using step 1 since we would apply step 3.2 to them only if both of them belong to the same DG. From step 1 we know that if the difference between the bases is $< \mathtt{d_1} \times \mathtt{CD_{Thres}}$, then they will belong to the same DG.

$$i.e.\ \text{if } |\mathtt{Base_2} - \mathtt{Base}| < \mathtt{d_1} \times \mathtt{CD_{Thres}} \tag{15}$$

then, both the accesses will belong to this DG.

The difference between the bases from eq.(13) and eq.(5) is

$$\mathtt{Base_2} - \mathtt{Base} = \sum_{\mathtt{j}=1}^{\mathtt{n}} \mathtt{E_j} \times \prod_{\mathtt{x}=\mathtt{j}+1}^{\mathtt{n}} \mathtt{n_j} \tag{16}$$

We know from eq.(14) and eq.(6) that:

$$\mathtt{d_1} = \mathtt{e_1} = \mathtt{C_1} \times \prod_{\mathtt{m=2}}^{\mathtt{n}} \mathtt{n_m} \tag{17}$$

Now, substituting eq.(16) and eq.(17) in eq.(15) we get:

$$\text{If } \sum_{\mathtt{j=1}}^{\mathtt{n}} \mathtt{E_j} \times \prod_{\mathtt{x=j+1}}^{\mathtt{n}} \mathtt{n_j} < \mathtt{C_1} \times \prod_{\mathtt{m=2}}^{\mathtt{n}} \mathtt{n_m} \times \mathtt{CD_{Thres}} \tag{18}$$

, then both the accesses will belong to this DG

$$\Rightarrow \frac{\mathtt{E_1}}{\mathtt{C_1}} + \frac{\mathtt{E_2}}{\mathtt{C_1} \times \mathtt{n_2}} + \cdots + \frac{\mathtt{E_n}}{\mathtt{C_1} \times \prod_{\mathtt{m=2}}^{\mathtt{n}} \mathtt{n_m}} < \mathtt{CD_{Thres}} \tag{19}$$

(which will be true in most cases since $\mathtt{C_1}$ and $\mathtt{E}$s are small positive numbers < 5 and $\mathtt{n_2} \cdots \mathtt{n_n}$ are relatively large, and $\mathtt{CD_{Thres}}$ is 5 in our experiments.)

Hence, both these accesses represented by $\mathtt{MemAddr(Base, d)}$ and $\mathtt{MemAddr(Base_2, e)}$ from the binary will belong to the same DG.

First, let us see what the bounds for the induction variable from source would be in the presence of the second access as well. We know that accesses from source do not access out of bounds in correct programs. We have seen that the bounds for each induction variable ($\mathtt{i_k}$) only considering the first access is $\frac{(\mathtt{n_k} - \mathtt{B_k})}{\mathtt{C_k}}$ as shown in eq.(8). Now considering that the second access does not access out of bounds we get:

$$\mathtt{C_k} \times \mathtt{i_k} + \mathtt{B_k} + \mathtt{E_k} < \mathtt{n_k} \tag{20}$$

$$\text{Rearranging the terms, } \mathtt{i_k} < \frac{(\mathtt{n_k} - (\mathtt{B_k} + \mathtt{E_k}))}{\mathtt{C_k}} \tag{21}$$

The difference between the bounds calculated from eq.(8) and eq.(21) is $\frac{\mathtt{E_k}}{\mathtt{C_k}}$

Now let us apply algorithm 3 to both these accesses.

We know that $\mathtt{Base_{diff}} = \sum_{\mathtt{j=1}}^{\mathtt{n}} \mathtt{E_j} \times \prod_{\mathtt{x=j+1}}^{\mathtt{n}} \mathtt{n_j}$. By dividing it with $\mathtt{gcd(d_k, e_k)} = \mathtt{d_k}$ (since $\mathtt{d_k} = \mathtt{e_k}$ in our case) repeatedly in loop and keeping the remainder of it for the next iteration we recover $\mathtt{x_k}$s of the form $\lfloor \frac{\mathtt{E_k}}{\mathtt{C_k}} \rfloor$ as long as $\mathtt{C_k}$s are factors of $\mathtt{E_k}$s. By subtracting $\mathtt{x_k}$s from the already present guesses of the induction variables we get $\mathtt{g_k} = \frac{\mathtt{C_{(k-1)}} \times \mathtt{n_k}}{\mathtt{C_k}} - \frac{\mathtt{E_k}}{\mathtt{C_k}}$. Many of the $\mathtt{E}$s will be zeroes, hence we will not make adjustment to many bounds, however we will make adjustment to the bounds that have small constant $\mathtt{E}$s in their terms. Further, it is good to note that the term we subtract using algorithm 3 is equivalent to the difference of the bounds as shown above.

It is important to note at this point that by subtracting from the already guessed bounds, we are making sure that the second access which accesses a few extra elements in some dimensions does not run into the higher dimension of the first access. This is very important because if we do not make this adjustment we will have extra dependencies from binary which will prevent parallelization and by subtracting the extra from bounds we will not see those spurious dependencies. Also it is important to note that the new guess we have for the bounds is also higher than or equal to the actual bounds of the loop.

5.4 Step 4: Create the Worklist

In this step we create a worklist with DGs that have accesses with remaining constraints so that we can apply step 5 on them to guess the upper bounds for the remaining induction variables. After step 3 we have upper bound constraints for

all the induction variables in the memory addresses other than the ones that correspond to the highest dimension in the write accesses. We need a method to guess the upper bound on these induction variables as well. This method is step 5. Hence, we now create a worklist with all DGs in which there is an induction variable for which we do not have an upper bound guess as yet. These would be the highest dimension induction variables since we do not have guesses for those after step 3. This worklist will enable us to work on only those DGs that have remaining constraints.

5.5 Step 5: Work on Inter-group Constraints

In this step we look at all DGs in the worklist created in step 4 (recall that these DGs have induction variables for which we have no guesses as yet) and solve for this DG not overlapping with the immediately following DG in the FullList. While creating DGs we assumed that each DG corresponds to a non-overlapping array region. Hence, it is required that different DGs do not overlap with each other; else this would generate false dependencies from binaries. Solving this generates further guesses on the remaining induction variables. These guesses are called inter-group constraints.

The formal method for solving that $\texttt{DG}_\texttt{i}$ from worklist does not overlap with $\texttt{DG}_{(\texttt{i+1})}$ (the immediately following DG in the FullList of DGs) is presented in algorithm 4, we describe it briefly here. For every $\texttt{DG}_\texttt{i}$ that has constraints remaining we substitute the guesses for all induction variables other than the highest one in all its memory expressions and require that this be less than the lowest base in $\texttt{DG}_{(\texttt{i+1})}$. Solving the above constraint we can obtain an higher bound for the highest induction variable. We then choose the minimum of the present guess and the already present minimum guess for that induction variable. This way we ensure that all our guesses are respected.

Algorithm 4. Step 5: Algorithm for Inter-group constraints

Input: Worklist from step 4 and guesses $g_1, g_2, \cdots, g_n$ from step 3.2
Output: Final guesses for bounds $g_1, g_2, \cdots, g_n$
for all DG_i in worklist after step 4 **do**
  for all $MemAddr(Base, d)$ in DG_i **do**
    $Base_{low}$ = Lowest Base from $DG_{(i+1)}$ in FullList
    $g_{11} = \lfloor \frac{Base_{low} - Base - (\sum_{j=2}^{n} d_2 * g_2)}{d_1} \rfloor$
    $g_1 = min(g_1, g_{11})$
  end for
end for

Intuition for step 5: Now that we have presented an algorithm for calculating the bounds on the highest induction variable, let us apply this to an access from source code, to show that our method guesses the value for the highest induction variable that is $\geq$ to the actual bound on that induction variable.

In step 3 we assumed we were working with loop nests comprising of the following induction variables $(\mathtt{say}, \mathtt{i}_1, \mathtt{i}_2, \cdots \mathtt{i}_n)$ and array accesses $\mathtt{A}[\mathtt{C}_1 \times \mathtt{i}_1 + \mathtt{B}_1] \cdots [\mathtt{C}_n \times \mathtt{i}_n + \mathtt{B}_n]$ in the loop, and the size of array A is $[\mathtt{n}_1][\mathtt{n}_2] \cdots [\mathtt{n}_n]$. Let this access belong to $\mathtt{DG}_i$.

First, let us recollect the guesses for all induction variables except the highest induction variable from step 3. One of the guesses we would have made for induction variable $\mathtt{i}_k$ (where $k \in$ [2,n]) is $\frac{\mathtt{C}_{(k-1)} \times \mathtt{n}_k}{\mathtt{C}_k}$ (eq.(9)). Hence, the final guess after step 4 will be equal to or lower than this guess.

Next, let us assume that there is an access to array B in the same loop belonging to $\mathtt{DG}_{(i+1)}$, *i.e.* the immediately following DG in the FullList. If this second array B is laid immediately after A in the binary, then $\mathtt{Base}_B$ will be at least:

$$\mathtt{Base}_B = \mathtt{Base}_A + \prod_{j=1}^{n} \mathtt{n}_j \text{ (this term is the size of A)} \tag{22}$$

Let us assume that all accesses corresponding to B belong to $\mathtt{DG}_{(i+1)}$. The lowest address of $\mathtt{DG}_{(i+1)}$ will be $\mathtt{Base}_B$.

Next, we apply the method in algorithm 4 for solving $\mathtt{DG}_i$ not overlapping with $\mathtt{DG}_{(i+1)}$ from source to derive the guess for $\mathtt{i}_1$ and then verify that this guess is correct. For doing so we must substitute our guesses for all the induction variables except the highest dimension induction variable in the expression of memory address A and this must be less than $\mathtt{Base}_B$. The expression for memory address A obtained by substituting the intra-group guesses eq.(9) in eq.(4) is:

$$\mathtt{Base}_A + \sum_{j=1}^{n} \mathtt{B}_j \times \prod_{x=j+1}^{n} \mathtt{n}_x + \mathtt{C}_1 \times \prod_{x=2}^{n} \mathtt{n}_x \times \mathtt{i}_1 + \sum_{j=2}^{n} \mathtt{C}_j \times \prod_{x=j+1}^{n} \mathtt{n}_x \times \left(\frac{\mathtt{C}_{j-1} \times \mathtt{n}_j}{\mathtt{C}_j} - 1\right) \tag{23}$$

The only unknown in eq.(23) is $\mathtt{i}_1$. This must be less than $\mathtt{Base}_B$ (from eq.(22)). Hence,

$$\begin{aligned} \mathtt{Base}_A + \sum_{j=1}^{n} \mathtt{B}_j \times \prod_{x=j+1}^{n} \mathtt{n}_x + \mathtt{C}_1 \times \prod_{x=2}^{n} \mathtt{n}_x \times \mathtt{i}_1 + \sum_{j=1}^{n} \mathtt{C}_j \times \prod_{x=j+1}^{n} \mathtt{n}_x - \sum_{j=2}^{n} \mathtt{C}_j \times \prod_{x=j+1}^{n} \mathtt{n}_x \\ \leq \mathtt{Base}_A + \prod_{j=1}^{n} \mathtt{n}_j \end{aligned} \tag{24}$$

Rearranging the terms we get,

$$\mathtt{i}_1 \leq \frac{\prod_{j=1}^{n} \mathtt{n}_j - (\mathtt{C}_1 \times \prod_{x=2}^{n} \mathtt{n}_x + \sum_{j=1}^{n} \mathtt{B}_j \times \prod_{x=j+1}^{n} \mathtt{n}_x)}{\mathtt{C}_1 \times \prod_{x=2}^{n} \mathtt{n}_x} \tag{25}$$

Further,

$$\mathtt{i}_1 \leq \frac{\mathtt{n}_1}{\mathtt{C}_1} - 1 - \frac{\mathtt{B}_1}{\mathtt{C}_1} - \frac{(\sum_{j=2}^{n} \mathtt{B}_j \times \prod_{x=j+1}^{n} \mathtt{n}_x)}{\mathtt{C}_1 \times \prod_{x=2}^{n} \mathtt{n}_x} \tag{26}$$

$$i.e.\ \mathtt{i}_1 \leq \frac{\mathtt{n}_1}{\mathtt{C}_1} - 1 - \frac{\mathtt{B}_1}{\mathtt{C}_1} - (\Delta) \tag{27}$$

The remaining values are small since the constant C's and B's are small and the sizes of arrays in affine code are generally large.

Hence, the guess for i_1 will be:

$$g_1 = \lfloor \frac{(n_1 - B_1)}{C_1} - 1 \rfloor \tag{28}$$

As seen before from source we require that the array expression must not exceed the size of the array dimension. Hence the highest dimension array expression $(\mathtt{C}_1 \times \mathtt{i}_1 + \mathtt{B}_1)$ must not exceed the highest dimension $(\mathtt{n}_1)$.

$$i.e.\ \mathtt{C}_1 \times \mathtt{i}_1 + \mathtt{B}_1 < \mathtt{n}_1 \tag{29}$$

Rearranging the terms $\mathtt{i}_1 < \frac{(\mathtt{n}_1 - \mathtt{B}_1)}{\mathtt{C}_1}$

Hence, the maximum value i_1 can take is $\frac{(n_1 - B_1)}{C_1} - 1$ and this is what we get by solving the equations from binary.

We have now seen that the algorithm 4 to calculate the bounds on the highest dimension induction variable yields a limit on it that is the true limit on it even from source code.

At the end of step 5, we now have made best guesses for all induction variables in the loop that appear in a memory address. If there is an induction variable that does not appear in any memory access, then we just assume that it can take any value since we have no way of determining its bounds. This does not hurt our method and is reasonable since even from source if an induction variable does not appear in any of the memory addresses present in the loop it could take any value at run-time and this would be legal.

For array accesses that came from dynamically allocated memory we apply the same algorithm described above. It is important to note that all ds in the MemAddr(Base, d) expression would be loop invariant symbols rather than constants. In many cases the memory expression we recover from binary code for these accesses will be of the form

$$\mathtt{Base} + x_1 \times x_2 \cdots x_n \times i_1 + x_2 \times x_3 \cdots x_n \times i_2 + \cdots + x_n \times i_n \tag{30}$$

where all the xs and Base are loop invariant quantities. By applying the algorithm to such an access we guess that the bound on i_k is $x_{(n-1)}$. We then check that the actual bounds are less than this loop invariant quantity (this check would succeed) before executing the parallel version of the loop.

Now that we have constraints on all the induction variables, we calculate the distance vectors and take parallelizing decisions for this loop assuming these as loop bounds. We then clone this loop and run the parallel version of the loop when the run-time checks for all induction variables succeed; else we run the serial version of the loop. Since we check at run-time that the loop bounds that we have guessed are actually correct we will always be conservatively correct. Please note that using the distance vector method to parallelize is our implementation method, one may use any parallelizing decision algorithm including polyhedral methods.

6 Implementation-SecondWrite

In this section we describe the binary rewriting infrastructure, SecondWrite Kotha et al. (2010); O'Sullivan et al. (2011); Anand and et. al. (2013) used for this research and how the automatic parallelizer interacts with rest of the system.

Architecture of Binary Rewriter called SecondWrite is presented in figure 2. SecondWrite's custom binary reader and de-compiler modules translate the input x86 binary into the intermediate representation (IR) of the LLVM compiler. LLVM is a well-known open-source compiler Lattner and Adve (2004) developed at the University of Illinois, and is now maintained by Apple Inc. LLVM IR is language and machine independent. Thereafter the LLVM IR produced is optimized using LLVM's pre-existing optimizations, as well as our enhancements, including automatic parallelization. Our new algorithm is implemented within this static

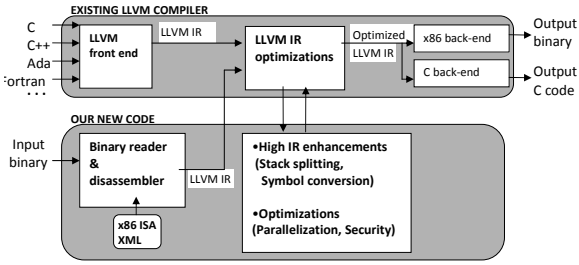

Fig. 2. SecondWrite

affine automatic parallelizer. Finally, the LLVM IR is code generated to output x86 code using LLVM's existing x86 code generator.

Currently SecondWrite rewrites x86 binaries from both Linux and Windows. It successfully rewrites binaries coming from source totaling over 2 million lines of code, including all of the SPEC2006 benchmarks. Real world programs such as the apache web server (230K+ LOC), Lynx browser (135K+ LOC) and MySQL (1.7M LOC) are also successfully rewritten. Rewritten benchmark binaries on average run 10% faster than highly optimized input binaries, and 45% faster than unoptimized input binaries because of the existing optimizations in LLVM not including parallelization.

SecondWrite is able to rewrite binaries without relocation information Smithson et al. (2010). SecondWrite implements various mechanisms O'Sullivan et al. (2011); Anand and et. al. (2013) to obtain an intermediate representation which contains features like procedure arguments, return values, types, high-level control flow, symbols and aggregate data structures. SecondWrite also employs extra mechanisms to safely handle indirect calls and indirect branches Smithson et al. (2010). It employs alias analysis frameworks present in LLVM to discover all the possible target procedures at indirect callsites, given by the points-to set of the operand in indirect call instruction. An edge is added from the indirect call-site to all its possible target procedures. Indirect branches are mostly present due to jump tables in the binary. Procedure boundary determination techniques are devised to limit the possible branch targets within the current procedure and extra control flow edges are added corresponding to the possible targets determined by alias analysis. If one of the target is outside procedure boundary, it is handled as an indirect call.

The algorithm presented in section 5 can be implemented in any static or dynamic binary rewriter as long as symbol recognition and induction variable analysis is implemented in the system.

7 Results

We use "-O3" optimized binaries from gcc-4.3 and gfortran-4.3 as input to SecondWrite, which includes the new algorithm proposed in this paper within a static

affine parallelizer. The static affine automatic parallelizer, that is in SecondWrite works on LLVM IR. We build a source automatic parallelizer by feeding it LLVM IR generated from *clang* LLVM (2007) (a C language front-end for llvm) for the 'C' benchmarks and LLVM IR generated using the *dragonegg* LLVM (2009) plugin (a plugin that integrates the LLVM optimizers and code generator with GCC) for the FORTRAN benchmarks. The LLVM IR fed to the stand-alone automatic parallelizer contains array location and dimension information, hence the source parallelizer uses it to take parallelization decisions. We run all the binaries on the AMD Opteron(TM) processor 6212 and present results.

In this section we present our results on parallelizing binaries from SPEC2006 and OMP2001 using our new algorithm. First, we introduce our benchmarks. Second, we present the speedups we have from source and binary. For the binary numbers, we present results for speedups both with and without the new algorithm. Third, we present the actual number of affine loops that are parallelized from the binary with and without the algorithm. We measure speedups by measuring the clock time to run the programs on 1 thread and 8 threads.

Table 1. Description of Benchmarks

Benchmark	*Language*	*# LOC*	*Suite*	*Benchmark*	*Language*	*# LOC*	*Suite*
swim	Fortran	275	OMP2001	quake	C	1151	OMP2001
bwaves	Fortran	680	SPEC2006	libquantum	C	2605	SPEC2006
mgrid	Fortran	789	OMP2001	milc	C	9575	SPEC2006
lbm	C	908	SPEC2006	cactus	Fortran + C	59827	SPEC2006

First, table 1 lists the 8 affine benchmarks that we present our results on. Our source and binary parallelizers correctly parallelize every benchmark from both the benchmark suites; however do not give any speedup on the remaining benchmarks since those benchmarks do not contain affine rich regions. We have picked only the affine rich benchmarks from the SPEC2006 and OMP2001 benchmark suites. We manually profiled every benchmark belonging to both the benchmark suites and after examining the hot regions classified benchmarks as affine or not affine. We present our results on all the affine benchmarks discovered from both the benchmark suites. The benchmarks *swim*, *mgrid* and *quake* belong to the OMP2001 benchmark suite and *bwaves*, *lbm*, *libquantum*, *milc* and *cactus* belong to the SPEC2006 benchmark suite. These benchmarks range from 275 to 59,827 lines of code as shown in table 1.

Second, figure 3 presents the speedup for 8 threads from source and binary for each of the benchmarks w.r.t the gcc "-O3" compiled single thread version of the benchmark. There are three bars for each benchmark; (i) the first bar is the speedup of the benchmark from source code for 8 threads; (ii) the second bar is the speedup of the binary for 8 threads without the new algorithm using only the theory presented in Kotha et al. (2010) and (iii) the third bar is the speedup of the binary for 8 threads using the new algorithm presented in this paper. We observe that *swim*, *bwaves*, *mgrid*, *quake*, *milc* and *cactus* gain significant speedups when the new algorithm presented in this paper is present in the static affine binary parallelizer.

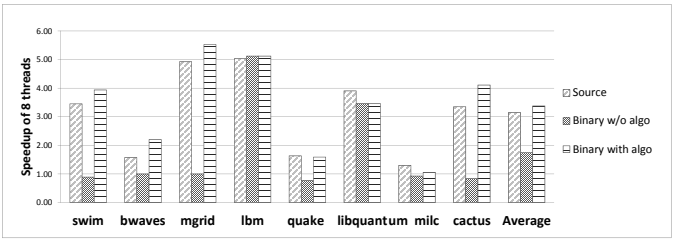

Fig. 3. Speedup of 8 threads for the affine benchmarks from SPEC2006 and OMP2001

The significant affine loops in these benchmarks have run-time determined loop bounds and hence using our new algorithm we are able to parallelize these loops that were not parallelized using the theory developed before. The benchmarks *lbm* and *libquantum* do not have any difference in the speedups with and without the algorithm. The reason being; (i) in *lbm*, the loops bounds are statically known and hence the theory in Kotha et al. (2010) is sufficient to parallelize the affine loops in it and (ii) in *libquantum* the loops are single dimensional with a write to one single dimensional memory accesses. These loops can be parallelized without the new algorithm and hence we see a speedup in *libquantum* even without the new algorithm. Overall the average speedup for 8 threads for the 8 benchmarks from binaries increases from 1.75X to 3.38X with the addition of the new algorithm. Our binaries run slightly faster than source since SecondWrite is able to rewrite "-O3" binaries to run 10% faster than the input binaries.

Table 2. Number of loops parallelized with and without the new algorithm

Benchmark	# loops w/o algo	# loops with algo	*Benchmark*	# loops w/o algo	# loops with algo
swim	6	18	quake	7	9
bwaves	0	1	libquantum	18	18
mgrid	0	6	milc	37	43
lbm	4	4	cactus	112	126

Third, table 2 presents the number of loops that are parallelized from the binary with and without the new algorithm. We observe that in the benchmarks *lbm* and *libquantum* the number of loops parallelized with and without the algorithm do not change. The reasons for this have been explained earlier. In *swim*, *quake*, *milc* and *cactus*, a number of loops are parallelized even when the new algorithm is not present in the static affine binary parallelizer; however, these loops are small and do not contribute to the run-time of the benchmark. Hence, these loops do not result in a speedup from 8 threads for these benchmarks. We make this comparison to show that it is not the number of loops that are parallelized that matter, but it is important to parallelize the run-time intensive loops that can be parallelized by our new algorithm.

8 Discussion and Future Directions

In this section we describe few salient aspects of our algorithm choice and alternate ideas that can be tested in the future.

8.1 Choice of a Heuristic Based Method

In the algorithm presented in this paper, one observes that we are solving a system of equations using a set of constraints to obtain loop bounds. We also observe that the number of constraints we have are not sufficient to solve for definite solutions for loop bounds. In this scenario, there are two possible methods to obtain a solution: (i) using a linear systems of equation solver that gives all possible solutions; and (ii) using a heuristic based solution relying on assumptions about loop structure and memory accesses, which gives one solution.

We choose a heuristic based approach over an equation solver for the following reasons: (i) it gives only one definite solution that can be used to insert run-time checks in the code and execute the parallel version only if the check succeeds; and (ii) it arrives at a solution in linear time complexity. One way of looking at our heuristic based algorithm is that it picks the one solution from all the possible solutions (that can be obtained using a solver) making assumptions on the loop structure most amenable to parallelization. Hence, even though there are many other solutions, this is the one that is mostly likely correct and going to yield from parallelization. Further, using our method we obtain the solution in linear time as against in exponential time complexity using a solver.

8.2 Future Directions

The present algorithm is two-dimensional; *i.e.* if the guess is wrong then at run-time it uses the fall-back solution and executes the serial version of the loop. In the future the following ideas can be used to enhance it.

1. If a run-time checks fails, then the loop bounds for that loop can be written to a log file and in future they can be used to parallelize the loop. That way in future with some run-time feedback, the algorithm to parallelize affine loops can be enhanced further.
2. In the present algorithm, we make a lot of assumptions to arrive at one set of loop bounds. In future, we will look into refining our assumptions to arrive at a few possible loop bounds and then use run-time feedback to use a different one in the next execution if this set fails in this run. This will provide fall-back solutions beyond serial execution.
3. In the present algorithm, we do not include any mechanism for user feedback. In future, we envision a system where the user can explicitly turn off parallelization of certain loops it they know that it would not be as profitable.

References

Anand, K., et al.: A compiler level intermediate representation based binary analysis and rewriting system. In: Proceedings of the 8th ACM European Conference on Computer Systems (2013)

Dasgupta, A., Dasgupta, A.: Vizer: A framework to analyze and vectorize intel x86 binaries (2003)

Franke, B., O'boyle, M.: Array recovery and high-level transformations for dsp applications. ACM Trans. Embed. Comput. Syst. (2003)

Yang, J., Soffa, M.L., Skadron, K., Whitehouse, K.: Feasibility of dynamic binary parallelization (2011)

Kotha, A., Anand, K., Smithson, M., Yellareddy, G., Barua, R.: Automatic parallelization in a binary rewriter. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (2010)

Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on CGO (2004)

LLVM, clang: a C language family frontend for LLVM (2007), `http://clang.llvm.org/`

LLVM, DragonEgg - Using LLVM as a GCC backend (2009), `http://dragonegg.llvm.org/`

Maslov, V.: Delinearization: an efficient way to break multiloop dependence equations. In: Proc. the SIGPLAN 1992 Conference on Programming Language Design and Implementation, pp. 152–161 (1992)

Nakamura, T., Miki, S., Oikawa, S.: Automatic vectorization by runtime binary translation. In: Proceedings of the 2011 Second International Conference on Networking and Computing (2011)

O'Sullivan, P., Anand, K., Kotha, A., Smithson, M., Barua, R., Keromytis, A.D.: Retrofitting security in cots software with binary rewriting. In: Proceedings of the 26th International Information Security Conference (2011)

Pradelle, B., Ketterlin, A., Clauss, P.: Polyhedral parallelization of binary code. ACM Trans. Archit. Code Optim. (2012)

Smithson, M., Anand, K., Kotha, A., Elwazeer, K., Giles, N., Barua, R.: Binary rewriting without relocation information. Technical report, University of Maryland, College Park (2010)

Wang, C., Wu, Y., Borin, E., Hu, S., Liu, W., Sager, D., Ngai, T.-F., Fang, J.: Dynamic parallelization of single-threaded binary programs using speculative slicing. In: Proceedings of the 23rd International Conference on Supercomputing, ICS 2009 (2009)

Yardimci, E., Franz, M.: Dynamic parallelization and mapping of binary executables on hierarchical platforms. In: Proceedings of the 3rd Conference on Computing Frontiers (2006)