Affine Parallelization using Dependence and Cache analysis in a Binary Rewriter

Aparna Kotha, *Student Member, IEEE,* Kapil Anand, *Student Member, IEEE,* Timothy Creech, Khaled ElWazeer, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua, *Member, IEEE,*

Abstract—Today, nearly all general-purpose computers are parallel, but nearly all software running on them is serial. Bridging this disconnect by manually rewriting source code in parallel is prohibitively expensive. Hence, automatic parallelization is an attractive alternative.

We present a method to perform automatic parallelization in a binary rewriter. The input to the binary rewriter is the serial binary executable program and the output is a parallel binary executable. The advantages of parallelization in a binary rewriter versus a compiler include: (i) applicability to legacy binaries whose source is not available; (ii) applicability to library code that is often distributed only as binary; (iii) usability by end-user of the software; and (iv) applicability to assembly-language programs.

Adapting existing parallelizing compiler methods that work on source code to binaries is a significant challenge. This is primarily because symbolic and array index information used by existing parallelizing compilers to take parallelizing decisions is not available from a binary. We show how to adapt existing parallelization methods to binaries without using symbolic and array index information to achieve equivalent source parallelization from binaries.

Results using our x86 binary rewriter called SecondWrite are presented in detail for the dense-matrix and regular *Polybench* benchmark suite. For these, the average speedup from our method for binaries is 27.95X vs 28.22X from source on 24 threads, compared to the input serial binary. Super-linear speedups are possible due to cache optimizations.

In addition our results show that our method is robust and has correctly parallelized much larger binaries from the SPEC 2006 and OMP 2001 benchmark suites, totaling over 2 million source lines of code (SLOC). Good speedups result on the subset of those benchmarks that have affine parallelism in them; this subset exceeds 100,000 SLOC. This robustness is unusual even for the latest leading source parallelizers, many of which are famously fragile.

1 Introduction

Since 2004 semiconductor trends show that the astonishing improvements in clock speeds have come to an end. However, improvements in silicon area per Moore's law, are still being realized. As a natural consequence, microprocessor vendors such as Intel and AMD have turned to multi-core processors to remain competitive. To fully utilize these multi-core processors, we must run parallel software on them.

One way to obtain parallel software is to manually rewrite serial code to parallel. This can be done either by using parallel language directives such as OpenMP to implicitly specify parallelism using comments in high-level language programs. The other way to manually obtain parallel software is to write programs in an explicitly parallel manner. This is done using a set of APIs, such as MPI, POSIX compliant *pthreads* or Intel's TBB, to extend existing languages such as C, C++ and Fortran. Although the use of such explicitly parallel programming is increasing, the adoption of manual parallel programming has been slowed by the following factors: (i) huge amounts of serial code represent most of the world's programs; (ii) rewriting code manually in parallel is time consuming and expensive; and (iii) a dearth of engineers trained in parallel programming and algorithms. For this reason, except for the most performance-critical code, it is not likely that most of the world's existing serial code will ever be rewritten in parallel.

The other way to obtain parallel software is to automatically parallelize serial source in a parallelizing compiler. This overcomes the above-mentioned drawbacks of manually rewriting parallel code. Indeed, since the introduction of parallel machines in the early 1970s, many strides have been made in parallelizing compiler technology. Most efforts to date have focused on parallelism in loops, primarily in regular, dense-matrix codes. In particular, techniques have been developed for parallelizing loops with array accesses whose indices are affine (linear) functions of enclosing loop induction variables [1]. This work is particularly interesting since most scientific and multimedia codes are affine and the maximum run time is

- *A. Kotha, K. Anand and K. ElWazeer have graduated with a PhD from the Department of ECE, University of Maryland, College Park, MD, 20742. Email:{akotha,kapil,khaled}@umd.edu*
- *T. Creech and M. Smithson are PhD students with the Department of ECE, University of Maryland, College Park, MD, 20742. Email:{tcreech,msmithso}@umd.edu*
- *G. Yellareddy has graduated with a masters from the Department of ECE, University of Maryland, College Park, presently working with Intel Corporation. Email:greeshma@intel.com*
- *R. Barua is an Associate Professor with the Department of ECE, University of Maryland, College Park, MD, 20742. E-mail:barua@umd.edu*

spent in these loops. Hence parallelizing these loops can result in significant speedups.

In this paper we propose methods to implement automatic parallelization inside a binary rewriter, instead of a compiler. A binary rewriter is a software tool that takes a binary executable program as input, and generates a transformed executable as output. In our case, the input code will be serial, and the output will be parallel code.

Parallelization in a binary rewriter has several advantages over a compiler: (i) Applies to third-party and legacy code for which no source is available, either because the developer is out of business or the code is lost; (ii) Works for binaries that contain hand-coded assembly-language code; (iii) Works for library code for which no source is available; and (iv) Can be used by an end-user to tune a generic distributed input binary to an output binary customized for the particular platform he/she is executing it on. Platform parameters that can be used include total cache size and cache line size.

The above advantages argue that *it is useful to have automatic parallelization in a binary rewriter*, despite compiler implementation being possible.

Parallelization of affine loops is more effective when cache effects are considered in parallelization decisions along with memory and scalar dependencies. Early work in automatic parallelization explored calculating dependencies in affine loops in the form of dependence or distance vectors [2] [3] [4] [5]. These vectors were then used to decide the most profitable loop nests to be parallelized. However, researchers soon realized that studying dependencies alone was not sufficient to obtain maximum performance from affine loops. Applying loop transformations such as loop interchange, loop fusion, loop fission, etc. increased performance tremendously [6] [7] [1] [8]. This increase in performance can be explained by noting that interchanging loop dimensions (for example) in a loop nest changes the access patterns of the arrays – this can change both the granularity of parallelism and the cache locality of the loop. Different interchanged orders of loop dimensions generally have very different run-times and hence it is important to choose the correct ordering of loops. Further, fusing different loop nests may increase the reuse across memory accesses and decrease the run-time of the loop. Hence, it is important to study the cache behavior in affine loops and take decisions about the loop transformations that need to be applied to a particular loop nest.

In our studies, we have found that applying loop transformations on affine kernels can lead to a 3X improvement in the serial run-time because of cache optimizations alone. Benefits are even greater when parallelization is done as well. This improvement in serial and parallel run-time is because cache reuse is maximized in the transformed loop. Hence, it is important to study cache effects on affine loops in any automatic parallelizer.

In this paper we propose a method to implement a *cache reuse model* in a binary rewriter building on the work to calculate dependence vectors to parallelize affine loops from binaries [9]. We also present solutions to many of the practical considerations that arise in implementing parallelization and cache analysis in a real binary rewriting framework.

Our approach to automatic parallelization and cache optimizations of affine loops from binaries is **not** to invent entirely new parallelization methods, but to adapt ideas from existing compiler methods to a binary rewriter. This adaption is not trivial, since binary rewriters pose challenges not present in a compiler, including primarily, the lack of high-level information in binaries. Parallelizing compilers rely on high-level information such as symbolic information, for identifying arrays, affine function indices, and induction variables; for renaming to eliminate anti and output dependencies; and for pointer analysis to prove that memory references cannot alias, allowing their parallel execution. Binaries lack symbolic information, making all these tasks more difficult. *A central contribution of this paper are parallelization methods in a binary rewriter that can work effectively without using any symbolic or array index information.*

Of course, we recognize that parallelizing affine programs is only one step towards the goal of parallelizing all programs, albeit an important one. Many programs have non-affine parallelism, and others have little or no parallelism. This work should be seen as what it is: a first successful attempt to parallelize binaries using affine analysis, rather than the last word. We hope to open up a new field of research with this significant step.

2 Related Work

This section lists related work in the following subsections.

2.1 Static binary auto-parallelization

Kotha et.al [9] and Pradelle et.al [10] are the only methods we are aware of that have done affine automatic parallelism from binaries. This journal paper is an extended version of our work presented in Kotha et.al. [9] that statically parallelizes binaries by using dependence information determined from binaries. A list of enhancements compared to [9] is submitted separately along with this paper. (Note to reviewers: We will replace this with a summary of differences in the final version.)

[10] automatically parallelizes binaries by feeding the binary intermediate form of loops only to the polyhedral compiler. First, our work in [9] preceded it. Further, our methodologies are better than [10] in the following ways: (i) [10] does not raise the entire binary

to an intermediate form like we do, greatly limiting the scope of parallelized loops. Our experimentation has showed that for recognizing induction variables that have been spilled to memory, advanced value set analysis like we have implemented in our binary rewriter and described in [11] is essential. This analysis needs to be applied to the entire binary and not just selective parts of it. Hence, though [10] works on small kernels, its scope of scaling to larger programs is limited; (ii) [10] relies on heuristics to be applied to the intermediate form to make it acceptable to the polyhedral compiler in terms of what it expects affine code to look like, whereas our method is built upon compiler theory. A heuristic based method that looks for code patterns is not universal and could be limited to one compiler or one optimization level. They do not provide any proof that their methods are universal. Our methods are based on compiler theory; hence we can parallelize binaries from different compilers and optimization levels; (iii) [10]'s results are limited to small kernels, whereas our results show that our methods scale well to benchmarks from SPEC2006 and OMP2001 that are 2-3 orders of magnitude larger; (iv) [10] generates parallel code that contains breakpoints before and after parallel code in order to redirect control flow to parallel code and back via context switches that is very expensive. In our binary rewriter no context switches are required; and (v) [10] does not incorporate any methods to undo compiler optimizations like we do (presented in the supplementary material), without which the scope of parallelization from any binary affine parallelizer is limited.

2.2 Dynamic binary auto-parallelization

Dynamic automatic parallelization techniques present in literature are Yardimci et.al [12], Wang et.al [13] and Yang et.al [14]. All the three methods suffer huge run-time overheads from dynamic analysis. Further, they do not optimize affine loops whereas our method does.

2.3 Affine based source auto-parallelizers

A large volume of research on automatic affine parallelization methods from source code exists; too many to list in full. A small sampling of the developed compilers include Polaris [15], SUIF [8], [7], [6], and pHPF [16], PROMIS [17], Parafrase-2 [18] and PLUTO [19]. All these automatic parallelizing compilers parallelize code from source, unlike our method. Our method builds on these existing methods, but has significant differences allowing it to work on binaries.

2.4 Distance and Direction Vector Calculation

Affine parallelism has required solving systems of linear diophantine equations [3] to calculate distance vectors. Various techniques have been proposed in literature to solve these equations. These include the Greatest Common Divisor (GCD) test [4], [2], Banerjee's inequalities [2], Single Index and Multiple Index Tests [3], [5], Multidimensional GCD [2], the delta test [20] and the omega test [21]. We adopt these tests proposed for source code to our binary automatic parallelizer. We have presently implemented the GCD, Single and Multiple Index tests to solve the linear diophantine equations that we recover directly from binaries.

2.5 Cache optimizations using binary rewriting

There has been some prior work in studying cache behavior from binaries and applying optimizations to decrease their runtime. Nethercote et.al [22] does dynamic binary instrumentation and uses hardware counters to measure the miss rate of caches. Using this cache miss rate information, they insert memory prefetch instructions into binaries to improve its cache performance. Weidendorfer et.al [23] builds on [22] and uses the dynamic cache profile information to perform optimizations such as array padding and blocking. Our method is different from these methods in two ways: (i) our method uses affine analysis to analyze and transform binaries whereas [22] [23] do not; and (ii) the methods in [22] [23] are dynamic and incur run-time overhead from instrumentation, analysis and transformation, whereas our method, which is static, has no such overheads.

2.6 Cache optimizations in auto-parallelizers

Many source parallelizers such as [24] [25] [26] use a *cache reuse metric* to transform and parallelize affine loops. Wolf et al [24] presents a method to do an ad-hoc search of loop transformations and calculates the *cache reuse metric* for each sequence of loop transformations and selects the best transformation order. McKinley's algorithm [25] improves on [24] and presents an algorithm to intelligently search for the best loop transformation. We adapt [25] to binaries since it uses the traditional model, that matches our implementation; although our techniques can be used with the polyhedral model as well. Bondhugula et al [26] also uses the *cache reuse metric* in [25] within the polyhedral model. Another reason we use [25] is that it avoids an exhaustive search of transformations; instead it presents a method to intelligently build the most efficient loop structure using the results of *cache reuse metric* calculation. We extend McKinley's algorithm to use strip-mining (well studied in affine literature) to further improve the results.

3 Dependence Analysis from Binaries

The greatest challenge in parallelizing binaries is in calculating dependence vectors. We first show how this is done from source code, and then consider how the same can be done from binary code.

3.1 From Source

To parallelize affine loops (source-code loops containing array accesses whose indices are affine (linear) functions of enclosing loop induction variables) from source, distance vectors are calculated using the symbolic and array information and they represent the dependencies present in the loop. These are then used to determine the loop nest levels best suited for profitable parallelization. The methods used to calculate distance vectors are well documented in affine literature and are described in detail in the supplementary section [1] [2] [3] [4] [5].

3.2 From Binary

It is important to observe that all the source code methods for calculating dependence rely on source code features; in particular, array declarations (including number of array dimensions and size of each), as well as index expressions of array accesses (such as $i+2$ and $2j-3$ in $A[i+2][2j-3]$.) *Unfortunately binary code lacks this information.* In production-level stripped binaries, no symbol tables are present, making it hard to know how many arrays there are, or their locations, dimension count and sizes. Nor is it apparent from looking at machine instructions containing registers and memory locations what the array index expressions are. *Lacking this information, source code methods cannot be applied to binary code.*

This section presents our method of doing dependence analysis from low-level code obtained from binary. The analysis does not need array declarations or index expressions, and works for production-level stripped binaries. The basic approach of our method is **not** to try to recreate such information (which is hard, and in some cases undecidable). Rather our approach sidesteps the need for array declarations and index expressions by using a method tailored to low-level code like that in binary code.

To illustrate the method, a source-code fragment and one of its possible binary code is shown in figure 1. The binary is shown in pseudo-code for comprehensibility, but actually represents machine code. Other binaries are also possible, but we will be able to illustrate the general principles of our method with this example. The binary code assumes that the array is laid out in row-major form, with the address of A[i,j] being computed as:

$$\&\mathtt{A}[\mathtt{i},\mathtt{j}] = \mathtt{Base} + \mathtt{i} * \mathtt{size\_j} + \mathtt{j} * \mathtt{elem\_size} \quad (1)$$

where `elem_size` is the size of an individual array element, and `size_j` is the size of the second array dimension, both in bytes. We present equations in row-major format to understand our techniques, but in no way are these techniques affected if the code is arranged in a column-major format. Further, the binary parallelizer does not need to know if the compiler generated code in row-major or column-major format. It only looks at the memory accesses from binary code and takes parallelizing decisions.

To derive equations directly from a binary, the following intuition is helpful: *it is a simple proof to show that for any affine array access, its address variable is provably always an induction variable in its immediately enclosing loop*. Of course, it is usually a *derived* induction variable [27], derived from the basic induction variables like `i` and `j` in the source[1].

We know that the address of every affine access in source is a derived induction variable. In the binary code in figure 1, `addr_reg` is the address register, which is an induction variable since it came from an affine access in source.

For each induction variable `addr_reg` that is used as an address for a memory reference in the binary, the theory in [9] shows that `addr_reg` is always expressible as:

$$\mathtt{addr\_reg} = \mathtt{Base}_{\mathtt{outer}} + \mathtt{num\_i} * \mathtt{step}_{\mathtt{i}} + \mathtt{num\_j} * \mathtt{step}_{\mathtt{j}} \quad (2)$$

Generalizing this to an n-dimensional loop yields:

$$\mathtt{addr\_reg} = \mathtt{Base}_{\mathtt{outer}} + \sum_{\mathtt{k}=1}^{\mathtt{n}} \mathtt{num\_k} * \mathrm{step}_{\mathtt{k}} \quad (3)$$

where `num_k` is the current number of iterations of the $\mathtt{k}^{\mathtt{th}}$ loop dimension, and $\mathtt{Base}_{\mathtt{outer}}$ and $\mathtt{step}_{\mathtt{k}}$ are constants recoverable from the binary using the theory presented in the supplementary material as well. In particular $\mathtt{step}_{\mathtt{k}}$ represents the constant that multiplies the induction variable of loop `k` when the affine address is linearized; further $\mathtt{Base}_{\mathtt{outer}}$ represents the constant base address from which all addresses of this array are calculated.

We must first check that all memory accesses in the loop are induction variables, for this loop to have been an affine loop in source. We only work on those loops and recover the linearized address expressions for every memory access. These equations are important since they will help us derive the *cache reuse metric*.

Array Memory Dependence Analysis from binary Although binaries lack array declarations, the method in [9] (also presented in the supplementary material) shows how dependence analysis can be done for memory references that came from source arrays in a manner that is nearly as powerful as source. The method shows that after discovering equations 3 for every memory references in a loop, they are considered in pairs; then using linear algebraic methods such as GCD, Delta and Omega tests dependence information between all the memory references is obtained. The dependence information is in the form of distance or direction vectors; these can be analyzed to take

1. A variable is a basic induction variable if its only update in the loop is a constant increment (called its step) in every iteration. A derived induction variable d is of the form $d = c_1 * i + c_2$, where i is a basic or derived induction variable with step s; hence d too is an induction variable with step $c_1 * s$.

Source Code

```
for i from lb_i to ub_i
    for j from lb_j to ub_j
       A[i,j] = A[i,j] + 1
    end for
end for
```

Binary Code

```
1       reg_lb_i ← lb_i
2       reg_ub_i ← ub_j
3       i' ← lb_i * size_j                    --(E)
4       reg_ub'_i ← ub_i * size_j
5  loopi:reg_lb_j ← lb_j
6       reg_ub_j ← lb_j
7       j' ← lb_j * elem_size
8       addr_reg ← Base + i' + j'             --(B)
9       reg_ub_addr ← Base + i' + ub_j * elem_size
10 loopj:load reg ← [addr_reg]
11      reg ← reg + 1
12      store [addr_reg] ← reg
13      addr_reg ← addr_reg + elem_size       --(A)
14      CMP addr_reg ≤ reg_ub_addr            --(C)
15      Branch if true to loopj
16      i' ← i' + size_j                      --(D)
17      CMP i' ≤ reg_ub'_i                    --(F)
18      Branch if true to loopi
```

Fig. 1: Example showing source code and it's binary code. The marked statements are related to induction variables and used by the theory in [9] to discover affine accesses.

decisions about loop parallelization. The method does not recreate arrays or their index expressions from binaries. Rather it uses a mathematical formulation on induction variables that are used to reference memory to show when the same or different references across loop iterations may alias to the same memory location, leading to a loop-carried dependence.

Once loop parallelization decisions are taken, our binary rewriter modifies the input binary to produce a multi-threaded output binary with loop iterations assigned among 'n' threads. It tries to load balance and execute approximately the same number of iterations on each thread. If the number of iterations are not an exact multiple of the threads, then a few threads will have one more iteration than the others.

4 Characterizing Cache Reuse

In this section we describe the calculation of the *cache reuse metric* first from source and then present our technology to adapt it to binaries. The cache reuse metric from source code is calculated using the index expressions available in source whereas from binary it is calculated using the linearized multi-dimensional equations recovered for every access from binary.

4.1 From source

This section overviews the strategy used to calculate $\mathrm{LoopCost}(l)$ *(or* $\mathrm{LC}(l)$*)*, which is defined by McKinley [25] as the total number of cache lines that will be accessed by the affine accesses in the *entire loop nest* when the dimension l is interchanged to the innermost position. It is worth noting that $\mathtt{LC}(l)$ does not measure the reuse directly. Instead $\mathtt{reuse} = \mathtt{Number\ of\ cache\ lines\ accessed\ assuming\ all\ misses} - \mathtt{LC}(l)$. The goal is that the final loop ordering should maximize reuse, which is the same as minimizing $\mathtt{LC}(l)$.

$\mathtt{LC}(l)$ is widely used in all types of affine analysis such as in traditional affine literature [24] and in the polyhedral framework [26]. Once this model is available, any candidate transformation's footprint can be calculated. For brevity, we overview the method here; details are in [25].

```
for i from lb_i to ub_i
  for j from lb_j to ub_j
   X[i]+ = β × A[j,i] × Y[j]
  end for
end for
```
(a) Original loop in gemver

```
for j from lb_j to ub_j
  for i from lb_i to ub_i
   X[i]+ = β × A[j,i] × Y[j]
  end for
end for
```
(b) Loop to maximize reuse

Fig. 2: Loop to illustrate Cache Reuse Algorithm

An example helps motivate how loop interchange can help performance by using $\mathtt{LC}(l)$. Figure 2(a) shows an example code in C; figure 2(b) shows the same code with the loops interchanged. Without loss of generality, assuming that C has row-major storage, the interchanged loop has better cache performance than the original. To see why, consider that the original code accesses different cache lines of array A in successive iterations of the inner loop, leading to no spatial reuse for array A. In contrast, the rewritten loop reuses the same cache line of A in successive iterations of the inner loop, leading to a smaller cache footprint and better run-time. Here, from the definition of LoopCost, $\mathtt{LC}(i) < \mathtt{LC}(j)$; hence suggesting that there is maximum reuse when loop i is innermost.

4.1.1 Construction of Reference Groups

To calculate $\mathtt{LC}(l)$ it is not correct to simply add the number of cache lines accessed by each reference; sharing between affine references must be accounted for. For example, $\mathtt{A}[i]$ and $\mathtt{A}[i+1]$ have nearly identical footprints; we should count the cache lines of only one of them. McKinley's method thus defines the concept of reference groups of affine memory references – intuitively, references are in the same reference group if they are "nearby" in the memory layout. It thereafter uses any one member's cache lines to represent the entire group. More formally, references $\mathtt{Ref}_1$ and $\mathtt{Ref}_2$ are said to belong to the same reference group with respect to a loop nesting level if there is either temporal or spatial reuse between them, detected as follows:

- There is **temporal reuse** between $\mathtt{Ref}_1$ and $\mathtt{Ref}_2$, if (a) they have a non-loop-carried dependence (*i.e.* , a dependence within the same loop iteration); or (b) they have a loop-carried dependence which at this nesting depth has a small constant d as distance

vector component ([25] has found $d < 10$ to work well), and all other entries in it are zero.

- There is **spatial reuse** between Ref_1 and Ref_2 if they refer to the same array and differ by at most a constant d0 in the last subscript dimension of the array indices, where d0 is less than or equal to the cache line size in terms of array elements. All other subscripts must be identical.

4.1.2 Calculating $LC(l)$ in terms of cache lines

Here we describe how $LC(l, Ref_k)$ is calculated for each reference group Ref_k at each nesting level l. We take one data reference from each reference group – we have already described that any reference in the reference group is representative of its cache behavior – and calculate the contribution towards $LC(l)$. Assume that we are working with a loop nest of the form $L = \{l_1, ..., l_n\}$, and $R = \{Ref_1, ..., Ref_m\}$ contains representatives from each of the reference groups. Each Ref_k has array indices of the $\{f_{k1}(i_1, .., i_n), ..., f_{kn}(i_1, .., i_n)\}$ form and its contribution to $LC(l)$ is decided based on the category it belongs to. The three categories Ref_k can belong to in loop dimension l are:

- **Loop invariant** - if the subscripts of the reference do not vary with this loop nesting level l, then it requires only one cache line for all iterations of this loop dimension *i.e.* if none of the f_k values vary with i_1, where i_1 is the induction variable associated with loop dimension l, then $LC(l, Ref_k) = 1$. These references are loop invariant and have temporal locality.
- **Consecutive** - if only the last subscript dimension (the column) varies with this loop nest and the coefficient multiplying this induction variable is $const_k$, then it requires a new cache line every $\frac{CACHE\_LINE\_SIZE}{(const_k * size\_of\_elem)}$ iterations. If we call this ITERS_IN_CACHE_LINE, then a total of $\frac{trip_l}{ITERS\_IN\_CACHE\_LINE}$ cache lines will be accessed for this loop nest (where $trip_l$ is the trip count of this loop dimension l). These references are consecutive and have have spatial locality. In other words, $LC(l, Ref_k) = \frac{trip_l}{ITERS\_IN\_CACHE\_LINE}$, if f_{kn} varies with i_1 and no other f_k varies in i_1.
- **No Reuse** - if the subscripts vary in any other manner, then the array reference is assumed to require a different cache line every iteration, yielding a total of $trip_l$ number of cache lines *i.e.* $LC(l, Ref_k) = trip_l$, otherwise.

Next $LC(l)$ is calculated as follows; and thereafter can be used in a variety of cache optimizations:

$$LC(l) = (\sum_{k=1}^{m} LC(l, Ref_k)) \prod_{h \neq 1} trip_h \quad \forall (l \in [1 : n]) \quad (4)$$

Adding up the $LC(l, Ref_k)$ for all reference groups gives the total number of cache lines that are required when this loop is executed once in the innermost position of the loop nest. The total numbers of cache lines accessed when this loop is the innermost loop is obtained by multiplying the sum by the trip counts of all other loops in the nest other than this one. This is representative of the total cache lines accessed by this loop when it is the inner most loop in this loop nest. If trip counts are not recoverable from the binary, we just use 10 (a constant) to represent it in the formula. This is a standard technique in compilers and performs well.

4.2 From binary

Both the steps in the cache reuse estimator above – deriving reference groups and calculating their footprint – rely on array index expressions denoted above as f_k. *Since these expressions are not available in a binary, the source-code reuse model above cannot be directly used on a binary*. Using theory in section 3, our binary rewriting framework recovers eq.(3) for every affine access present in a loop directly from a binary and these equations are analyzed to calculate the *cache reuse metric*.

Next, we show how the source code method can be applied without using the array index expressions only relying on the induction variables that are recovered from the binary.

4.2.1 Generation of reference groups

Assume that there are two references Ref_1 and Ref_2 in a loop of nesting depth n in a binary. Then we can express their addresses using eq.(3) as shown in section 3 as follows:

$$addr\_reg_1 = Base_{outer1} + \sum_{k=1}^{n} num\_k * step_{1k} \quad (5)$$

$$addr\_reg_2 = Base_{outer2} + \sum_{k=1}^{n} num\_k * step_{2k} \quad (6)$$

Recall that the base and step values are constants. We can now replace the earlier source-level method for calculating reference groups with a binary-level formula. As earlier, two references Ref_1 and Ref_2 belong to the same reference group if there is either temporal or spatial reuse between them.

From a binary, the above two references Ref_1 and Ref_2 have temporal reuse between them (and therefore are in the same reference group) if they access the same memory location after d iterations of loop nest k. We can check this condition by checking if all the step1s are equal to the corresponding step2s and the difference of $Base_{outer1}$ and $Base_{outer2}$ is a multiple of $step_k$ and the quotient is d. There might be other more complex cases when after d iterations the two references access the same memory; however we found that using the above check covers most of the common cases and our benchmarks perform well.

Another condition by which two references Ref_1 and Ref_2 have a temporal reuse between them is if there is a non-loop carried dependence between them, *i.e.* the references access the same memory location in every iteration. This is detected from a binary if all the steps and $Base_{outer}$ are equal to the corresponding

ones in the other reference. However, this check is the subset of the previous check and hence checking only the previous case will serve both purposes. Hence, there is **temporal reuse** between two accesses $\texttt{Ref}_1$ and $\texttt{Ref}_2$ in a binary *if*:

- (a) the coefficients multiplying all the induction variables in eq 5 and eq 6 are the same for both the accesses; *i.e.* $\forall \texttt{l} \in [1:\texttt{n}]\ \texttt{step}_{11} = \texttt{step}_{21}\}$
- (b) the bases of both the accesses differ by a small multiple of the coefficient of the loop nesting level we are considering. *i.e.* , if we are considering loop nest k: (i) $(\texttt{Base}_{\texttt{outer1}} - \texttt{Base}_{\texttt{outer2}})\%\texttt{step}_{1\texttt{k}} = 0$; and (ii) $\frac{(\texttt{Base}_{\texttt{outer1}} - \texttt{Base}_{\texttt{outer2}})}{\texttt{step}_{1\texttt{k}}} \leq \texttt{d}$, where d is a small number (heuristically set at less than ten).}

Recall that there is **spatial reuse** between $\texttt{Ref}_1$ and $\texttt{Ref}_2$ and thus they belong to the same reference group from source *iff* the last subscripts differ by at most a constant d0 (a constant that places both the references on the same cache line) and all other subscripts are identical. Recall that the constant always rolls into the $\texttt{Base}_{\texttt{outer}}$ in the expression of the references from a binary and the coefficients multiplying the induction variables are rolled into the steps. Hence, we need to check that the $\texttt{Base}_{\texttt{outer1}}$ and $\texttt{Base}_{\texttt{outer2}}$ differ by less than the CACHE_LINE_SIZE. Further, we require all other subscripts to be identical in source and the last one to differ only by a constant, hence the coefficients multiplying induction variables are all equal. We can check this condition from a binary by ensuring that all the step1s are equal to the corresponding step2s. Hence, there is **spatial reuse** between two accesses $\texttt{Ref}_1$ and $\texttt{Ref}_2$ in a binary *iff*:

- (a) the coefficients multiplying all the induction variables are the same for both the accesses; *i.e.* $\forall \texttt{l} \in [1:\texttt{n}]\ \texttt{step}_{11} = \texttt{step}_{21}$ } and
- (b) the bases differ by less than the cache line size in bytes. *i.e.* $(\texttt{Base}_{\texttt{outer1}} - \texttt{Base}_{\texttt{outer2}}) < \texttt{CACHE\_LINE\_SIZE}\}$

4.2.2 Calculating $\texttt{LC}(\texttt{l})$ in terms of cache lines

Here we describe how $\texttt{LC}(\texttt{l}, \texttt{Ref}_\texttt{k})$ is calculated for each reference group at each nesting level directly from a binary; and then $\texttt{LC}(\texttt{l})$ is calculated from it. Just like from source, we take one data reference as a representation of each reference group and calculate its contribution towards $\texttt{LC}(\texttt{l})$. To do so, like from source, the following three categories are defined for reference groups, but the mathematical formulation is different. For example, if we consider loop nest k, an access is:

- **Loop invariant** - if the address expression from the binary does not contain a term for this loop nest, then it requires only one cache line for all iterations of this loop dimension. This condition corresponds to none of the $\texttt{f}_\texttt{k}$ values varying with $\texttt{i}_1$ from source since we know that the coefficients roll into the steps in a binary. In other words, if $\texttt{step}_{\texttt{kl}} = 0$, then $\texttt{LC}(\texttt{l}, \texttt{Ref}_\texttt{k}) = 1$. These references are loop invariant references and have temporal locality.
- **Consecutive** - if the address expression obtained from a binary has a multiplicative factor $\texttt{step}_{\texttt{kl}}$ associated with this loop nest level and $\texttt{step}_{\texttt{kl}}$ is less than the cache line size, then a new cache line is required every $\frac{\texttt{CACHE\_LINE\_SIZE}}{\texttt{step}_{\texttt{kl}}}$ iterations. If we call this ITERS_IN_CACHE_LINE, then a total of $\frac{\texttt{trip}_1}{\texttt{ITERS\_IN\_CACHE\_LINE}}$ number of cache lines will be accessed by this loop nest. This check is equivalent to our consecutive check in source, where we check that only the last subscript varies with this loop nest. In other words, $\texttt{LC}(\texttt{l}, \texttt{Ref}_\texttt{k}) = \frac{\texttt{trip}_1}{\texttt{ITERS\_IN\_CACHE\_LINE}}$, if $\texttt{step}_{\texttt{kl}} < \texttt{CACHE\_LINE\_SIZE}$ and $\texttt{ITERS\_IN\_CACHE\_LINE} = \frac{\texttt{CACHE\_LINE\_SIZE}}{\texttt{step}_{\texttt{kl}}}$
- **No Reuse** - if the affine expression is of any other form, then the array reference is assumed to require a different cache line every iteration, yielding a total of $\texttt{trip}_1$ number of cache lines accessed *i.e.* $\texttt{LC}(\texttt{l}, \texttt{Ref}_\texttt{k}) = \texttt{trip}_1$, otherwise.

Next similar, to the source formulation $\texttt{LC}(\texttt{l})$ is calculated as follows, completing our binary method:

$$\texttt{LC}(\texttt{l}) = (\sum_{\texttt{k}=1}^{\texttt{m}} \texttt{LC}(\texttt{l}, \texttt{Ref}_\texttt{k})) \prod_{\texttt{h} \neq \texttt{l}} \texttt{trip}_\texttt{h} \quad \forall (\texttt{l} \in [1:\texttt{n}])$$

This would be representative of the total cache lines accessed by this loop when it is the inner most loop in this loop nest.

The *intuitions* for the above equations in section 4.2 may seem familiar to readers, and indeed they are similar to those from source code. However, the *equations* are novel, since in previous work the corresponding equations are expressed in terms of source artifacts like original loop induction variables and arrays, neither of which are available in binaries. It is not difficult, *but certainly not obvious* to reformulate them in terms of binary artifacts such as address induction variables only, without using arrays. That is the primary contribution of section 4.2.

5 Forming an Optimization Strategy

In this section we describe how $\texttt{LC}(\texttt{l})$ calculated in section 4 from source or binary is used in McKinley's search strategy from [25] to take complex transformation decisions. We choose this search strategy (from traditional affine literature) to present proof of concept of how transformation orders can be decided in a binary. The same cache model can be used in a *Polyhedral* model for binaries as well.

McKinley's algorithm uses $\texttt{LC}(\texttt{l})$ to determine the ideal loop order for the different loops present in a particular loop nest. It then uses the dependence information to obtain the legal loop ordering (called Optimal order) closest to the ideal order. We then perform multi-level blocking of the loop nest to further increase reuse. These are further described below.

Determining the Ideal Loop Order Even though $\texttt{LC}(\texttt{l})$ does not directly measure reuse across outer

loops, it can be used to determine the loop permutation for the entire nest which accesses the fewest cache lines by relying on the observation *"If loop* `i` *promotes more reuse than loop* `j` *when both are considered for the innermost loop,* `i` *will promote more reuse than* `j` *at any outer loop position."* [25]. McKinley's algorithm thereafter orders the loops from outermost to innermost by decreasing values of their *LC(l)*. This order is the best theoretical order for this loop nesting which we call the ideal order.

Determining the Optimal Loop Order The optimal order of the loops is calculated using the the ideal order and loop dependencies by the following method. Start from the loop that is outer-most in the ideal order and check if it is legal to place it at the outer most position. If yes, place it and examine the next loop in the ideal order. If not examine the next outermost loop and check if it can be placed in the outermost position. Repeat until all loops have been placed.

Using the dependence information calculated from binaries as shown in section 3 and the *cache reuse metric* defined in section 4 for binaries, we can thus take complex loop transformation decisions on binary code.

Please note that we have used the above search strategy only as an implementation proof in our software; however, any search strategy used in source code can be used. More details of our implementation are presented in the supplementary section. We implement our affine binary automatic parallelizer within the ***SecondWrite*** binary rewriter [28] [11] [29].

We describe our strategies specific to binaries and algorithm used to parallelize loops with run-time dependent bounds in the supplementary section.

6 Results

We use "-O3" optimized binaries from GCC as input to *SecondWrite*, which includes our *dependence analysis* and *cache reuse model*. The source automatic parallelizer includes the exact same optimizations as *SecondWrite* and works on LLVM IR. Hence, we use LLVM IR from "llvm-gcc" or "clang" as the input to our source parallelizer, which we use for comparison with our binary parallelizer.

In this section we present results of our binary parallelizer for the benchmarks from *polybench* benchmark suite on the 24 core Xeon E7450 machine. We also have detailed results on the 8 core Xeon E5530 machine in the supplementary section. Further, results on the affine benchmarks from the SPEC2006 and OMP2001 benchmark suites are presented in the supplementary section.

Detailed results on the polybench benchmark suite are presented in figure 3. For each benchmark we present the speedup (with respect to 1 thread from source) in columns representing 1, 2, 4, 8, 16 and 24 threads. The rows represent the different configurations used to obtain the speedup. The five benchmarks, *2mm*, *3mm*, *gemver*, *gemm* and *doitgen* that benefit from cache analysis are presented on the left side. The remaining benchmarks are presented on the right side and the average is presented at the bottom right corner.

We first present the general trends in our results, followed by some salient results.

First, we observe that the basic parallelizer from binary using dependence analysis is as powerful as the source parallelizer using dependence analysis only. Actually, in most cases the speedup from binaries is slightly higher than the source parallelizer. The first two rows of the results for each benchmark in figure 3 show this result. The reason for this is that the binary parallelizer works on gcc "-O3" optimized binaries, raises them to LLVM intermediate form and then generates an output binary by applying all of LLVM's code optimizations. In effect, code optimizations from two compilers are applied to our binaries performing slightly better than the source parallelizer that includes code optimizations only from LLVM. For *e.g.*, we carefully analyzed *gessumv* which has the highest speedup from binary and observed that there was one less memory spill in the most time-intensive loop in the binary version resulting in 30% speedup. Further, we observe that *correlation* does not get parallelized from binary code. The reason is that *correlation* has two array accesses that each access the lower triangular and upper triangular elements of an array; however from binary code we conservatively assume that these two accesses could alias with each other since the loop bounds are coupled and the present linear algebraic methods implemented in our system are conservative in this case.

Second, we observe that our binary cache reuse methods significantly improve performance compared to a basic binary parallelizer. Looking at the binary results (shaded in light gray) of benchmarks that benefit from cache analysis in figure 3, we observe that: (i) the geomean speedup without cache analysis was 14.66X for 24 threads on BUZZ whereas it was 36.87X (2.5X better) with cache analysis. Super-linear speedups are possible with cache optimizations. We also observe that our binary methods can perform as well as source-level cache-reuse methods. Some of the speedups are higher than source with cache optimizations since they benefit from code optimizations from two compilers.

Third, we compare the results of our parallelizing compiler with PLUTO, a parallelizing compiler from source code based on the polyhedral method [19]. The geomean speedup of our benchmarks with PLUTO is 3.95X on BUZZ whereas the speedup from our automatic parallelizer with cache analysis is 8.31X on BUZZ. In affine literature many decision algorithms have been suggested each suited for some bench-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2349501, IEEE Transactions on Parallel and Distributed Systems

BENCHMARKS THAT BENEFIT FROM CACHE ANALYSIS							
Benchmark		1	2	4	8	16	24
2mm	Source w/o cache opt	1.00	2.09	3.94	7.95	14.34	22.98
	Binary w/o cache opt	1.21	2.42	4.64	9.13	17.63	26.01
	Source with cache opt	4.86	9.83	19.14	37.53	73.08	103
	Binary with cache opt	4.92	9.73	18.42	37.37	73.38	104.2
	Source with PLUTO	3.06	6.24	12.37	21.56	41.85	59.37
3mm	Source w/o cache opt	1.00	1.92	3.81	7.20	14.05	20.82
	Binary w/o cache opt	0.68	1.35	2.71	5.36	10.59	14.98
	Source with cache opt	3.99	7.62	15.84	30.88	60.03	85.09
	Binary with cache opt	3.99	7.97	15.50	31.16	59.84	85.34
	Source with PLUTO	0.66	0.85	0.86	0.85	0.84	0.84
gemver	Source w/o cache opt	1.00	1.95	3.55	3.88	4.85	5.75
	Binary w/o cache opt	0.97	1.76	2.81	3.68	3.90	4.64
	Source with cache opt	1.90	4.09	6.71	8.15	7.13	3.99
	Binary with cache opt	2.12	3.37	5.46	6.30	5.00	3.58
	Source with PLUTO	2.20	3.48	5.87	8.86	9.01	8.34
gemm	Source w/o cache opt	1.00	1.98	3.82	6.99	13.40	19.7
	Binary w/o cache opt	1.08	2.16	3.92	7.45	14.27	20.74
	Source with cache opt	2.61	5.21	10.37	20.23	38.49	54.32
	Binary with cache opt	2.61	5.22	9.97	20.29	38.49	54.42
	Source with PLUTO	1.32	2.32	4.31	8.24	15.60	22.12
doitgen	Source w/o cache opt	1.00	1.98	3.86	7.28	13.35	17.13
	Binary w/o cache opt	1.12	2.22	4.21	7.55	14.33	18.05
	Source with cache opt	2.62	5.19	10.18	19.78	32.22	39.01
	Binary with cache opt	2.55	5.07	9.83	18.32	34.04	39.31
	Source with PLUTO	1.21	2.44	5.03	7.78	13.28	16.01

BENCHMARKS THAT DO NOT NEED CACHE ANALYSIS							
Benchmark		1	2	4	8	16	24
atax	Source	1.00	2.20	4.36	4.21	4.01	3.36
	Binary	0.97	1.95	3.92	3.52	4.20	3.87
	Source with PLUTO	1.37	2.51	4.31	4.21	3.98	3.99
jacobi	Source	1.00	2.31	4.36	7.14	10.97	9.28
	Binary	1.07	2.42	4.85	7.68	8.62	6.98
	Source with PLUTO	0.35	0.35	0.35	0.35	0.35	0.35
bicg	Source	1.00	2.29	4.44	5.06	8.08	7.16
	Binary	0.94	1.96	3.60	4.61	4.67	6.45
	Source with PLUTO	1.01	1.70	2.82	2.47	2.42	2.37
gesummv	Source	1.00	1.67	2.61	3.44	1.82	1.71
	Binary	1.30	1.93	2.64	3.28	2.01	1.63
	Source with PLUTO	1.92	4.90	8.09	9.84	7.76	4.63
correlation	Source	1.00	1.15	1.87	2.85	2.54	1.85
	Binary	1.08	1.04	1	1.04	0.97	0.9
	Source with PLUTO	1.27	1.25	1.91	2.49	2.31	1.76
covariance	Source	1.00	1.06	1.53	2.37	2.20	1.64
	Binary	0.99	0.89	0.88	0.9	0.83	0.75
	Source with PLUTO	1.33	1.04	1.33	1.62	1.25	0.93
Geo Mean (all benchmarks)	Source	1.65	3.03	5.49	8.33	10.31	9.86
	Binary	1.72	2.88	4.69	6.60	7.94	8.31
	Source with PLUTO	1.25	1.87	2.89	3.64	4.06	3.95
Geo Mean (benchmarks that benefit from cache analysis)	Source w/o cache opt	1.00	1.98	3.80	6.47	11.18	15.62
	Binary w/o cache opt	0.99	1.94	3.57	6.32	10.83	14.66
	Source with cache opt	3.02	6.08	11.65	20.68	32.94	37.49
	Binary with cache opt	3.08	5.86	10.88	19.37	31.03	36.87

Fig. 3: Speedup on x86 E7450 with 24 threads (BUZZ) with respect to 1 thread from source

marks, however none of them perform well on all the benchmarks. We observe a similar effect here when we compare our decision algorithm to that present in PLUTO. PLUTO performs better than our algorithm on a few benchmarks and scales better on BUZZ for some benchmarks than our decision algorithm. However, on average our decision algorithm is better than PLUTO's on the benchmarks from *Polybench*. We do not suggest that PLUTO or the polyhedral model is less powerful; we merely wish to present results from PLUTO to show that our speedups are good compared to the best publicly available academic polyhedral compiler from source code.

Fourth, we present a study of the improvement in L1 cache miss rates with our cache optimizations in table 1 for the benchmarks that benefited from cache reuse analysis. The baseline source is generated using the LLVM based parallelizer on source code whereas baseline binary is generated by using the *SecondWrite* infrastructure on gcc "-O3" optimized binaries. We observe that the L1 cache misses are significantly reduced after cache optimizations are applied to candidate loops.

Benchmark	*Baseline Source*	*+ cache opt*	*Baseline Binary*	*+ cache opt*
2mm	28.6%	3.2%	33.3%	2.1%
3mm	33.3%	3.2%	33.3%	2.1%
gemver	9.9%	2.2%	10.8%	2.7%
gemm	33.2%	4.6%	24.9%	3%
doitgen	33.2%	3%	24.9%	2%
Average	27.64%	3.24%	25.44%	2.38%

TABLE 1: L1 cache miss rate with optimizations

Next, we discuss the trends we observe for some individual benchmarks from figure 3: (i) For the *2mm* benchmark, we observe that the speedup scales well with number of threads on BUZZ. The speedup from PLUTO is consistently lower than our decision algorithm with cache optimizations and higher than the speedup from our base parallelizer; however, it also scales very well with increasing number of threads; (ii) For the *gemver* benchmark, we observe that the speedups scales well till 8 threads and beyond that the speedups either stabilize or decrease. This can be be attributed to the fact that *gemver* works on single-dimensional arrays (hence, is not as time intensive as other benchmarks) which when divided among more than 8 threads is hurt by the synchronization overhead; (iii) For the *doitgen* benchmark, we observe that there is good scaling in speedup till 16 threads, beyond which the speedup does not scale well. Further, we observe that the parallelization achieved by PLUTO is close to our base parallelizer and our cache optimizations improve over these speedups.

7 Conclusions and Future Work

In this work we presented techniques to parallelize binary code especially focusing on affine loops. Our techniques are able to parallelize affine loops, transform loop nests to maximize cache reuse and also handle loops whose bounds are only known at runtime. We have presented techniques that show how source code parallelization techniques can be adapted to binary code when symbolic and array information is not available. Our results show that affine rich programs from the Polybench, SPEC 2006 and OMP 2001 benchmark suites have speedups from binary code are similar to those from source.

In future, this work must be explored in at least the following directions: (i) integrate the affine parallelizer with non-affine parallelization techniques to increase the scope of benchmarks parallelized; (ii)

integrate a more sophisticated decision algorithm to combine many more loop transformations. Many such algorithms have been presented in the source literature and can be adapted to binaries using some of the ideas that we have presented; (iii) A polyhedral compiler called Polly is under development within the LLVM infrastructure. Since our infrastructure builds over LLVM, in future we can use our techniques to feed Polly directly from binary code and present results; and (iv) understand how the cache optimization techniques presented can be applied to already parallel code coming from OpenMP/TBB/Cilk etc.

REFERENCES

[1] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[2] U. Banerjee, *Dependence Analysis for Supercomputing*. Boston: Kluwer Academic Publishers, 1988.

[3] U. Banerjee and U. of Illinois at Urbana-Champaign. Dept. of Computer Science, *Speedup of Ordinary Programs*, ser. UIUCDCS-R. Department of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[4] R. A. Towle, "Control and data dependence for program transformations." Ph.D. dissertation, Champaign, IL, USA, 1976.

[5] M. J. Wolfe, "Optimizing supercompilers for supercomputers," Ph.D. dissertation, Champaign, IL, USA, 1982.

[6] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Detecting coarse-grain parallelism using an interprocedural parallelizing compiler," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 1995.

[7] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the suif compiler," *Computer*, vol. 29, no. 12, pp. 84–89, 1996.

[8] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "Suif: an infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Not.*, vol. 29, no. 12, pp. 31–37, 1994.

[9] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua, "Automatic parallelization in a binary rewriter," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[10] B. Pradelle, A. Ketterlin, and P. Clauss, "Polyhedral parallelization of binary code," *ACM Trans. Archit. Code Optim.*, Jan. 2012.

[11] K. Anand, M. Smithson, K. ElWazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler level intermediate representation based binary analysis and rewriting system," in *European Conference on Computer Systems*, 2013.

[12] E. Yardimci and M. Franz, "Dynamic parallelization and mapping of binary executables on hierarchical platforms," in *CF '06: Proceedings of the 3rd conference on Computing frontiers*. New York, NY, USA: ACM, 2006, pp. 127–138.

[13] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T.-f. Ngai, and J. Fang, "Dynamic parallelization of single-threaded binary programs using speculative slicing," in *Proceedings of the 23rd international conference on SC*, ser. ICS '09.

[14] M. L. S. J. Yang, K. Skadron and K. Whitehouse, "Feasibility of dynamic binary parallelization," 2011.

[15] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Parallel programming with polaris," *Computer*, vol. 29, no. 12, pp. 78–82, 1996.

[16] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo, "An hpf compiler for the ibm sp2," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*.

[17] H. Saito, N. Stavrakos, S. Carroll, C. D. Polychronopoulos, and A. Nicolau, "The design of the promis compiler," in *CC '99: Proceedings of the 8th International Conference on Compiler Construction*.

[18] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors," *Int. J. High Speed Comput.*, vol. 1, no. 1, pp. 45–72, 1989.

[19] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2008.

[20] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 1991, pp. 15–29.

[21] W. Pugh, "The omega test: a fast and practical integer programming algorithm for dependence analysis," in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 1991, pp. 4–13.

[22] N. Nethercote and A. Mycroft, "The cache behaviour of large lazy functional programs on stock hardware," in *Proceedings of the 2002 workshop on Memory system performance*, 2002.

[23] J. Weidendorfer, M. Kowarschik, and C. Trinitis, "A tool suite for simulation based analysis of memory access behavior," in *In Proceedings of International Conference on Computational Science*, 2004.

[24] M. E. Wolf, D. E. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 29, 1996.

[25] K. S. McKinley, "A compiler optimization algorithm for shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, Aug. 1998.

[26] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[27] A. W. Appel and M. Ginsburg, *Modern Compiler Implementation in C*. Cambridge University Press, January 1998.

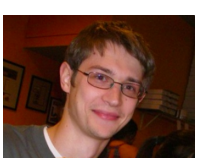
[28] P. O. Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. Keromytis, "Retrofitting security in cots software with binary rewriting," in *Proceedings of the 26th International Information Security Conference*, 2011.

[29] M. Smithson, K. Elwazeer, K. Anand, A. Kotha, and R. Barua, "Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness." in *WCRE*. IEEE, 2013, pp. 52–61.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2349501, IEEE Transactions on Parallel and Distributed Systems

Aparna Kotha Aparna is presently the Chief of Product Development at SecondWrite LLC, a startup aimed at commercializing binary rewriting technologies in the application performance monitoring and cyber-security space. Prior to joining SecondWrite, she received her doctorate in Computer Engineering from University of Maryland, College Park in Spring 2013 and bachelors in Electrical Engineering from Indian Institute of Technology, Madras in Spring 2006. Her field of interest is Computer Architecture and High Performance Computing. Her doctorate thesis was automatic parallelization of binaries and cache analysis of binaries.

Kapil Anand Kapil is currently a Co-founder and Chief Technology Officer at SecondWrite LLC. He earned a Bachelor of Technology degree in Electrical Engineering from the Indian Institute of Technology, Delhi in Spring, 2007 and a PhD in Electrical and Computer Engineering from University of Maryland, College Park in Summer, 2013. His research interests include binary rewriting, static program analysis and information-flow security. He received the Distinguished Dissertation Award from the Department of Electrical and Computer Engineering at University of Maryland, College Park for his research on binary rewriting techniques.



Timothy Creech Timothy Creech is currently a PhD student in the Department of Electrical and Computer Engineering at the University of Maryland, College Park. He obtained a B.S. in Computer Engineering from the University of Maryland in 2005. His current research interests are in runtime systems, tools, and operating systems for parallel machines. Tim's work is funded by a NASA Office of the Chief Technologist's Space Technology Research Fellowship.

Khaled Elwazeer Khaled ElWazeer is presently the Chief Architect at SecondWrite LLC. He received his PhD in Electrical and Computer Engineering at the University of Maryland College Park. Previous to his PhD, he got his bachelors and masters degrees from Cairo University in 2006 and 2009. His research interests include program analysis, programming languages, static analysis and reverse engineering.

Matthew Smithson Mr. Smithson attended Georgia Tech where he received his Bachelors degree in Computer Engineering.

He went on to work in the defense industry as an embedded systems software developer, later transitioning to malware research and reverse engineering. During this time, Mr. Smithson also completed the Masters Degree program in Computer Science from Johns Hopkins University.

Mr. Smithson currently serves as Mitsubishi Electric's Software Development Architect, where he designs software solutions for embedded, desktop, and web applications. In addition , he is currently pursuing a PhD in Computer Engineering from the University of Maryland, where he researches new static binary rewriting techniques.

Greeshma Yellareddy Greeshma is currently working at Intel Corporation as Technical Consulting Engineer. She received her MS degree in Electrical and Electronics Engineering from University of Maryland, College Park in 2013. Her research interests include parallelization, high performance computing, mobile and web technologies

Rajeev Barua Dr. Rajeev Barua is an Associate Professor of Electrical and Computer Engineering at the University of Maryland. He is also the Founder of CEO of SecondWrite LLC, which commercializes binary rewriting technology his research group developed at the university. He received his Ph.D. in Computer Science and Electrical Engineering from the Massachusetts Institute of Technology in 2000.

Dr. Barua is a recipient of the NSF CAREER award in 2002, and of the UMD George Corcoran Award for teaching excellence in 2003. He was a finalist for the Inventor of the Year Award in 2005 given by the Office of Technology Commercialization at the University of Maryland. Dr. Baruas work on the Raw Architecture at MIT ultimately led to a chip fabricated by IBM Corporation. His work on the Raw compiler has been incorporated by Tilera Corporation. Earlier, he received the President of India Gold Medal for graduating from the Indian Institute of Technology in 1992 with the highest GPA in the university that year. In 2013, his 1999 paper on ”Parallelizing Applications into Silicon” was selected among the most significant 25 papers in the first 20 years of the International IEEE Symposium on Field-Programmable Custom Computing Machines.

Dr. Barua’s research interests are in the areas of compilers, binary rewriters, embedded systems, and computer architecture. Recent work has tackled the problems of binary rewriting for security enhancement, real-time improvement, and lowering energy use. Earlier work has targeted compiler approaches to reliable software in embedded systems, memory allocation for embedded systems, compiling to VLIW processors, and compiling for multiprocessor and tiled architectures.