Automatic Parallelization in a Binary Rewriter

Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy and Rajeev Barua
Department of Electrical and Computer Engineering
University of Maryland, College Park, 20742
{*akotha,kapil,msmithso,greeshma,barua*}*@umd.edu*

***Abstract*—Today, nearly all general-purpose computers are parallel, but nearly all software running on them is serial. However bridging this disconnect by manually rewriting source code in parallel is prohibitively expensive. Automatic parallelization technology is therefore 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) compatibility with all compilers and languages; (ii) high economic feasibility from avoiding repeated compiler implementation; (iii) applicability to legacy binaries; and (iv) applicability to assembly-language programs.

Adapting existing parallelizing compiler methods that work on source code to work on binary programs instead is a significant challenge. This is primarily because symbolic and array index information used in existing compiler parallelizers is not available in a binary. We show how to adapt existing parallelization methods to achieve equivalent parallelization from a binary without such information. Preliminary results using our x86 binary rewriter called SecondWrite on a suite of dense-matrix regular programs including the externally developed Polybench suite of benchmarks shows an average speedup of 5.1 from binary and 5.7 from source with 8 threads compared to the input serial binary on an x86 Xeon E5530 machine; and 14.7 from binary and 15.4 from source with 32 threads compared to the input serial binary on a SPARC T2. Such regular loops are an important component of scientific and multi-media workloads, and are even present to a limited extent in otherwise non-regular programs.

***Keywords*-Automatic Parallelization; Binary Rewriting; Affine Dependence Analysis;**

I. Introduction

Since about 2004, semiconductor trends show that the astonishing improvements in clock speeds over the last few decades have come to end. However improvements in silicon area, as per Moore's law, are still being realized. As a natural consequence of these trends, microprocessor vendors have turned to multi-core processors to remain competitive. For example, Intel corporation has replaced its Pentium line of uniprocessors with the Intel *Core* processor family, virtually all of which have multiple cores. AMD corporation offers the Athlon Dual-core and Phenom Quad-core processors. By the end of 2009, multi-cores accounted for 100% of all new desktop and notebook processors [1]. The CPU road maps of both Intel and AMD show trends towards further multiple cores.

One way to obtain parallel software is to use parallel language directives such as OpenMP [2], [3] to implicitly specify parallelism using comments in high-level language programs. The other way to obtain parallel software is to write programs in an explicitly parallel manner. This is done using a set of APIs, such as MPI [4], posix complaint *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 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; (iii) dearth of engineers trained in parallel programming and algorithms; and (iv) parallel programming productivity per line of code is lower than for serial [5]. For this reason, except for the most performance-critical code, it is not likely that most of the world's existing serial code will be rewritten in parallel.

The other way to obtain parallel software is to automatically parallelize serial source code in a parallelizing compiler. Automatic parallelization overcomes the above-mentioned drawbacks of explicitly 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 [6]. This work is particularly interesting as most scientific and multi-media codes are affine and the maximum run time is spent in these loops. Hence parallelizing these loops can result in significant speedups.

In this paper we develop 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 outputs an improved executable as output. In our case, the input code will be serial, and the output will be parallel. *As far as we know, there are no existing methods for automatic parallelization in a static binary rewriter. Further, there are no existing binary automatic parallelization tools (static or dynamic) that perform affine based parallelization*. Parallelization in a binary rewriter has several advantages over parallelization

in a compiler:

- **Works for all compilers and languages** A parallelizer in a binary rewriter works for all binaries produced using any compiler from any source language. This compatibility is a huge practical advantage versus a compiler implementation.
- **No need to change software toolchains** A binary rewriter is a simple add-on to any existing software development toolchain. Developers and their companies, typically very resistant to changing the toolchains they are most familiar with, will not have to. This is important since many existing compilers do not perform automatic parallelization.
- **High economic feasibility** A parallelizer in a binary rewriter needs to be implemented only once for an instruction set, rather than repeatedly for each compiler.
- **Applies to legacy code** Legacy binaries for which no source is available, either because the developer is out of business, or the code is lost. can be parallelized using a binary rewriter. No compiler can do this.
- **Works for assembly-language programs** A binary rewriter, unlike a compiler can parallelize assembly code, regardless of whether it is part of the program with inlined assembly or all of it. Assembly code is used sometimes to write device drivers, code for multi-media extensions, memory-mapped I/O, and time-critical code portions.
- **Can perform platform-specific tuning** Since a binary rewriter can tune the output program for the particular platform it is executing on, it is possible to tune the same input executable differently for different platforms which share the same ISA, but may have widely different runtime costs. For example, we already choose the best barrier and broadcast mechanism for the end-user platform, and are investigating specific optimizations for the instruction set enhancements and instruction latencies of that platform.
- **Can be used by end user of software** Unlike compiler-provided parallelization which can only be done by the software developer, parallelization in a binary rewriter can be done by the end user of the system depending upon his or her specific needs, constraints and environment.

The above advantages argue that *it is useful to provide automatic parallelization in a binary rewriter*, despite compiler implementation being possible. By allowing automatic parallelization to be done on *arbitrary binaries from any source*, we hope to make this technology *universal, accessible, portable, customizable to the end-user's platform, and usable by any computer user, not just developers*. In this vision, our hope is that the parallelizing rewriter will be a transparent utility that is automatically triggered for all programs at install-time. This transparent rewrite may perform other services for the end-user in addition to parallelization such as serial platform-specific optimization and site-specific security.

Our approach to automatic parallelization is **not** to invent entirely new parallelization methods, but to investigate how best to adopt ideas from existing compiler methods to a binary rewriter. This adoption 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 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.*

On the flip side, binary rewriters also enjoy an advantage compared to a compiler: they have access to the entire program including library code. The need for separate compilation – an important practical requirement – has meant that commercial compilers typically have access to only one file at a time. For large programs with many files, this represents a tiny sliver of the total code. Whole-program parallelization is important since parallelization in one procedure may inhibit parallelization at containing or contained procedures, and moreover, parallel code blocks may be in different files. Currently we look at loop-level parallelism as most execution time is spent in loops.

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. Our collaborators and we are actively working on techniques to parallelize non-affine codes as well, and have seen early indications of very promising results. When that work is ready it will be separately published. 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.

Yardimci and Franz [7] is the only method we are aware of that has done automatic parallelism in a binary rewriter. However unlike our method their method focuses on non-affine parallelism, and is dynamic. We decided to statically parallelize since the complex and time-intensive affine parallelization analysis is infeasible dynamically at run-time. Further, in the future, we envision integrating an affine decision algorithm combining various loop transformation techniques such as loop interchange, loop fusion, loop fission, reductions, array privatization, loop reversal and others.

This paper is further arranged in the following way. Section II shows how dependence vectors can be calculated from a binary, contrasting the method from source code. Section III and IV show how aliasing problems are resolved and scalar dependencies detected, respectively, from a binary. Sections V and VI discuss how loop partitioning and code generation may be accomplished from binaries, respectively. Section VII describes our implementation strategy. Section VIII describes our experimental setup and results. Section IX presents the related work and section X presents conclusions and future work.

II. Calculating dependence vectors

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.

A. From source

This section overviews the strategy to calculate dependence vectors in the form of distance or direction vectors from affine loops (source-code loops containing array accesses whose indices are affine (linear) functions of enclosing loop induction variables). For example, if *i* and *j* are loop induction variables, then array accesses A[i] and A[2i+j+3][i-3j+7] are affine, whereas A[i/2] is not. We will present the techniques first from the source and then we will adapt it from a binary in section II-B. The source-level techniques reviewed in this section are well documented in the literature of affine loop parallelism.

To understand how parallelization can be done for affine-based loops, consider that dependencies between the instructions limit the parallelism that we can extract in code. For loops, loop-carried dependencies are the major inhibitors of parallelism, and occur when a loop iteration cannot be initiated before some previous set of loop iterations has completed. Just like scalar dependencies, loop-carried dependencies can be classified into three types: true, anti, and output loop-carried dependencies. Figure 1 shows examples of each type.

As in existing work, based on the formulation in [8], a dependence vector($\vec{D}$) for loops is defined as an n-dimensional vector, where n is the nesting depth of the loop. The most common formulation of a dependence vector is a *distance vector*, where each entry is the step of the loop dependence in that induction variable. For example, for the code in figure 1(a) and 1(c) the distance vector is $\vec{D}$ = (1,0), indicating that there is a dependence in steps of 1 along i, whereas there is no dependence along induction variable j. Conversely, in figure 1(b) the distance vector is $\vec{D}$ = (0,2), indicating that there is a dependence in steps of two along induction variable j, and no dependence along induction variable i.

Distance vectors are calculated using the GCD and Single Index Variable (SIV) test [8]. The linear system of equations is obtained from the symbolic array index expressions and array declarations present in source code. Other source techniques for calculating distance vectors are presented in Section IX. Further when the distance cannot be found or is not deterministic, we can represent the dependence in loops by direction vectors [9], a less precise formulation of dependence vectors.

B. From Binary

This section presents our method of doing dependence analysis from low-level code obtained from binary code, which does not contain any symbolic information or affine expressions. The analysis will be successful when the underlying access patterns are affine, even when the array indices needed for traditional dependence analysis for parallelization are absent, such as in binary code.

A source-code fragment and one of its possible binaries is shown in Figure 2. 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:

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

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 assume row-major accesses to understand our techniques , but in no way are these techniques going to be effected if the code was arranged in a column-major format.

To see how to parallelize the binary code, 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 [10], derived from the basic induction variables like i and j in the source[1].

Loops in a binary can be recognized by analyzing the control flow graph that we obtain from a binary. Every back edge in the control flow graph defines a loop [10]. We know that the address of every affine access in the body of the loop is a derived induction variable. In the binary code in figure 2, `addr_reg` is the address register, which must be an induction variable since it came from an affine access in source. Starting from this address induction variable `addr_reg`, we can define the following six special statements in the binary ((A) to (F)) for every address variable in a loop that is an induction variable. This six

[1]Basic induction variables are those which are incremented by a constant in every loop 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 $c_1 * s$.

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

    end for
end for
```
(a) True-loop carried dependence

```
for i from lb_i to ub_i
    for j from lb_j to ub_j
       A[i,j] += A[i,j+2] + 1

    end for
end for
```
(b) Anti-loop carried dependence

```
for i from lb_i to ub_i
    for j from lb_j to ub_j
       A[i,j] = .........
       A[i-1,j] = .........
    end for
end for
```
(c) Output-loop carried dependence

Figure 1. Loop-Carried Dependencies in Source Code

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
```

Figure 2. Example showing source code and it's binary code

statements will help us parallelize the binary, regardless of the exact binary code available:

- **(A) – Address variable increment** The rewriter searches for the increment of the address induction variable in the loop, and names it (A). See the example binary in figure 2 to find (A) to (F).
- **(B) – Address variable lower bound** The incoming value of the address induction variable (`addr_reg` in the example) is its lower bound; it is marked (B).
- **(C) – Address variable upper bound** The upper-bound comparison of the address variable for the loop-ending branch identifies the upper-bound of the address variable. It is searched for and marked (C).
- **(D) – Outer loop induction variable increment** We check if stmt (B)'s right-hand side value contains another induction variable. If it does, it is distinguished as the induction variable of the next-outer loop. In the example it is `i'`. The increment which reveals this induction variable is marked (D).
- **(E) – Outer loop induction variable lower bound** The incoming value of the outer loop induction variable (`i'` in the example) is its lower bound; it is marked (E).
- **(F) – Outer loop induction variable upper bound** The upper-bound comparison of the outer loop induction variable for the loop-ending branch identifies the upper-bound of the address variable. It is searched for and marked (F).

Statements (A) to (C) are for the inner loop; and (D) to (F) are for the outer loop, if present. For loops nested to depth three or more, additional statements can be identified (e.g. (G) to (I) and so on). These special statements *can be identified from almost any binary compiled from affine accesses*, regardless of its exact form. Recognizing statements (A) to (F) in the binary relies primarily on effective induction variable analysis, which is easy for registers in binaries. By the definition of an induction variable, once it is recognized, its increment (or set of increments) reveal the statements (A) and (D). The incoming values ((B) and (E)) immediately follow, as well as the exit conditions ((C) and (F)).

Our recognizer will recognize not only virtually all affine accesses written as such, **but also affine accesses through pointers**, since the pointers themselves will be induction variables. The only non-recognized case is when the constant increment of the induction variable is hidden by layers of indirection, e.g. when the constant is in a memory location, or when the induction variable is not virtual-register-allocated in the binary rewriter's intermediate representation, but we have found such cases to be extraordinarily rare.

Let us define the address variables's lower bound value (RHS of (B)) as `Initial_addr_reg`, defined as:

$$\texttt{Initial\_addr\_reg} = \texttt{Base} + \texttt{lb}_\texttt{i} * \texttt{size\_j} + \texttt{lb}_\texttt{j} * \texttt{elem\_size} \quad (2)$$

From this we can rewrite addr_reg as:

$$\texttt{addr\_reg} = \texttt{Initial\_addr\_reg} + \texttt{num\_i} * \texttt{size\_j} + \texttt{num\_j} * \texttt{elem\_size} \quad (3)$$

where num_i and num_j are the number of iterations on loop i and loop j respectively.

The special statements (A) to (F) are helpful since they will help us in doing the remaining tasks in the rewriter – (i) deriving dependence vectors; (ii) deciding which loop dimensions can be partitioned; and (iii) actually performing the partitioning to generate parallel code. These are done in turn below.

Deriving dependence vectors Next we aim to define the dependence vector between pairs of array accesses in the loop. To do so, we consider any two derived induction variable references in a loop (not necessarily the two in the code example above) with addresses addr_reg1 and

addr_reg2. Their expressions can be derived by substituting `Initial_addr_reg` in `addr_reg` above, yielding:

$$\texttt{addr\_reg}_1 = \texttt{Base}_1 + \texttt{lb}_\texttt{i} * \texttt{size\_j} + \texttt{num\_i} * \texttt{size\_j} + \texttt{lb}_\texttt{j} * \texttt{elem\_size} + \texttt{num\_j} * \texttt{elem\_size} \quad (4)$$

$$\texttt{addr\_reg}_2 = \texttt{Base}_2 + \texttt{lb}_\texttt{i} * \texttt{size\_j} + \texttt{num\_i}' * \texttt{size\_j} + \texttt{lb}_\texttt{j} * \texttt{elem\_size} + \texttt{num\_j}' * \texttt{elem\_size} \quad (5)$$

After deriving these equations, the next step is to calculate the distance vector (d1 , d2) associated with these accesses. Say that (num_i , num_j) and (num_i' , num_j') are the iterations where add_reg_1 and $addr\_reg_2$ alias to the same memory location, then by definition (num_i - num_i' , num_j - num_j') is the distance vector associated with these accesses [2]. Hence, to calculate this distance vector we need to equate the R.H.S of 4 and 5. The unknowns in the equation are num_i , num_j , num_i' and num_j'. We now have four unknowns and one equation. But we also have the following bounds for these unknowns , as they are the number of iterations of loop i and j.

$$0 \leq \texttt{num\_i}, \texttt{num\_i}' \leq \lfloor \frac{\texttt{ub}_\texttt{i} - \texttt{lb}_\texttt{i}}{\texttt{size\_j}} \rfloor \quad (6)$$

$$0 \leq \texttt{num\_j}, \texttt{num\_j}' \leq \lfloor \frac{\texttt{ub}_\texttt{j} - \texttt{lb}_\texttt{j}}{\texttt{elem\_size}} \rfloor \quad (7)$$

We derive these bounds from statements (B), (C), (E), (F). For loops nested with higher depths there will be statements (H) , (I), ... to determine the bounds. We now solve for the distance vectors using the equation and bound conditions. One of the following four conditions may happen:

- There is no solution to this equation in the given space. This means that the two addresses do not alias with one another. We add a distance vector of (0, 0) to this loop.
- There is a deterministic solution (X,Y) for (d1, d2). This means that the loop refer to the same memory address after X iterations of i and Y iterations of j. We add the constant (X,Y) to the distance vector of this loop.
- There are multiple deterministic solutions to this equation. Then we add all the deterministic solutions to the distance vectors of this loop.
- In all other cases, when there are countably many solutions or when we are unable to determine the solution, the direction vector added has elements per loop dimension that are a combination of $<$, $>$, = and * [9]depending on the dimension that is uncountable and the direction in which it is uncountable.

Traditional affine theory defines the (Greatest Common Divisor) GCD test [11], [8] , Banerjee test [12], Delta Array tests [13] and the Single Index Variable (SIV) and Multiple Index Variable (MIV) tests [12], [6] to solve the linear equations that we derive from source. We use the same techniques to solve the equations from low-level code. Multiple tests have been defined as the techniques evolved to more precise solutions in increasing order of complexity.

If the bounds of num_i and num_j are unknown we can still say something about these equations in the infinite space. But of course this is not always true. Hence, we have developed techniques specific to a binary in case of unknown bounds and these are presented in section III.

The techniques presented in this section are different from source in the following way:

- The equations to be solved are directly derived from the binary as against the symbolic array index expressions readily available from source but absent in binaries.
- From source we derive the distance vectors by solving each dimension separately, where as from binary we derive the equations equivalent to linearizing the array. We do this since there is no symbolic information in the binary to determine array bounds and dimensions. We have found that these techniques are nearly as powerful as source techniques on the Polybench benchmark suite. We have been able to discover the same dependence vectors from source as well as the corresponding binaries. In extremely rare cases, the dependence vectors from binaries are less precise than from source, but still conservative and correct. This impact is measured in section VIII.

III. Resolving aliasing uncertainties

In compiling source code, the compiler only looks for data dependencies between references to the same array. References to different arrays are assumed to never alias – for example, references A[x] and B[y] are assumed to never alias, regardless of the values of x and y. In non-array-bounds-checking languages like C, in the actual layout the two might actually alias if one of the arrays is accessed out-of-bounds, but all compilers presume this never happens. This is legitimate since in ANSI C, no layout assumptions are allowed. Hence compilers can (and do) legitimately ignore aliases between different accesses to different arrays.

However, a binary rewriter cannot easily disregard such false aliasing dependencies between different arrays. The reason is that, in a binary rewriter, the location and size of arrays is unknown, so we do not know where one array ends and another begins. It is incorrect to assume that the base of each array reference is a different array, since, as mentioned below, the base may include constant offsets in the array indices. For example, the source reference A[i+5] will have the constant base address of A[5] in the binary, which is not the start of an array. Of course we will not know if the address really represents the middle of one array or a different array. Left unsolved, this problem will lead to far more dependencies in the binary than in source, hurting parallelizability.

[2]Distance vectors need to be lexicographically positive, hence if (num_i - num_i') is negative then the distance vector is (num_i' - num_i , num_j' - num_j)

Fortunately it is possible to remove such false aliasing dependencies between references using optimizations in the binary rewriter. First, if $Base_1$ and $Base_2$ provably access different segments (e.g. one is the stack pointer plus a constant, and the other is an address in the global segment), then they access non-intersecting addresses, and any dependence between their references. Second, if $Base_1$ and $Base_2$ are different addresses in the same segment but the loop bounds are constants, then we use one of the tests (GCD , SIV , MIV) described above to compute if the two accesses are independent. Intuitively, when the base values differ by greater than the size of the arrays, these tests return to us that these two accesses are not dependent on each other. This represents the case when the address ranges of the references are non-intersecting, i.e. $Base_1 + lb_i * size\_j + lb_j * elem\_size > Base_1 + ub_i * size\_j + ub_j * elem\_size$ **or** $Base_2 + lb_i * size\_j + lb_j * elem\_size > Base_2 + ub_i * size\_j + ub_j * elem\_size$ is true. We can check these conditions when the lower and upper bounds of loops are known constants in the binary rewriter. If the ranges are non-intersecting, the false aliasing dependence can be deleted.

Non-Constant Loop Bounds If the loop bounds are not constants then we will not be able to use the last-mentioned technique to eliminate false dependencies. Although we will still be able to use Banerjee's inequalities in a limited number of cases to prove independence between different arrays, they do not always work. Hence, we have also developed techniques specific to binaries to help us prove independence between possibly different arrays when the loop bounds are unknowns.

To explain our strategy for non-constant loop bounds consider figure 3. Figure 3(a) shows the source code for a simple loop with three memory accesses: two to array A and one to array B. Figure 3(b) shows the corresponding binary code with accesses to three different addresses. The three address variables will have the following form from the theory developed in section II:

$$\texttt{addr1} = \texttt{Base}_1 + \texttt{lb}_\texttt{i} * \texttt{size\_j} + \texttt{num\_i} * \texttt{size\_j} + \texttt{lb}_\texttt{j} * \texttt{elem\_size} + \texttt{num\_j} * \texttt{elem\_size} \quad (8)$$

$$\texttt{addr2} = \texttt{Base}_2 + \texttt{lb}_\texttt{i} * \texttt{size\_j} + \texttt{num\_i}' * \texttt{size\_j} + \texttt{lb}_\texttt{j} * \texttt{elem\_size} + \texttt{num\_j}' * \texttt{elem\_size} \quad (9)$$

$$\texttt{addr3} = \texttt{Base}_3 + \texttt{lb}_\texttt{i} * \texttt{size\_j} + \texttt{num\_i}'' * \texttt{size\_j} + \texttt{lb}_\texttt{j} * \texttt{elem\_size} + \texttt{num\_j}'' * \texttt{elem\_size} \quad (10)$$

The above address expressions are identical except for different constant base values. These addresses are analyzed to form *Reference Groups*(RG) by comparing the values of $Base_1$, $Base_2$, $Base_3$. If the bases differ by less than a pre-determined constant threshold we group them together into an RG. In this example $addr1$ and $addr2$ will form one reference group (say RG1) and $addr3$ will belong to a different reference group (say RG2). Intuitively the idea will be that references in different RGs will be checked at run-time to see if they indeed alias using a low-cost loop-invariant range check; parallel code is executed only when no dependence is found at run-time. References within the same RG are assumed to likely access the same array with high probability, and hence are not checked at run-time for independence, since that check will rarely find independence.

To see what the run-time check looks like, for each reference group we calculate the lowest and highest address that this reference group can reference as a symbolic expression of unknown loop bounds. For example in RG1 the lower bound on the address will be $Base_1 + lb_i * size\_j + lb_j * elem\_size$ and the upper bound will be $Base_2 + ub_i * size\_j + ub_j * elem\_size$ as $Base_1$ will be surely less than $Base_2$. For RG2 the lower bound is $Base_3 + lb_i * size\_j + lb_j * elem\_size$ and upper bound is $Base_3 + ub_i * size\_j + ub_j * elem\_size$ as there is only one reference in RG2.

The code generated in this method is shown in figure 3(c). The loop is cloned for each of the two outcomes of the run-time check. The run-time code checks that every pair of RGs have non-intersecting address ranges. If the check succeeds then we execute the loop that obeys all dependencies within RGs but none between them; else we assume that all RGs are dependent on each other. More sophisticated versions of the check are possible and will be considered in the future. We will succeed on the run-time checks for a majority of the cases as we will be trying to prove independence between different arrays which is true. Further the number of run-time checks will be limited as in scientific codes the number of arrays accessed in the body of one loop are limited to three or four in most cases. Finally the run-time check is done outside loops, minimizing their overhead to near negligible levels.

IV. Scalar dependencies

A scalar dependence is present in a loop if a location is defined in one iteration of the loop and used in another iteration of the loop. Conceptually, detecting and handling scalar dependencies is similar in source code and binaries. One minor difference is that whereas the possibly dependent locations in source code are variables, in binaries they are registers and memory locations. Our scalar dependence analysis for binaries is outlined below.

We recognize register/scalar dependencies from a binary by analyzing def-use chains. All registers are checked to see if they are defined in one iteration and used in a later iteration. This is a check on def-use chains of registers, to see if the register is live at the exit block of the loop. Traditional data flow can be run on low-level code from binaries. We leverage this to check the presence of loop-carried register dependencies. We check for the presence of register dependencies at every loop depth as a certain dependence may be present at one depth and not at another loop depth. For example in the code in figure 4 *sum* has a

```
for i from lb_i to ub_i
   for j from lb_j to ub_j
       A[i,j] = ....
       .... = A[i,j+3]
       B[i,j] = ....
   end for
end for
```

(a) Source Code

```
Loop_i : ........
   Loop_j : .......
       *addr1 = ....
       .... = *addr2
       *addr3 = ....
   end Loop_i
end Loop_j
```

(b) Binary Code

```
if(non-intersecting ranges between
all pairs of RGs)
   Loop with no dependencies
   between all pairs of RGs
else
   Loop with dependencies
   between all pairs of RGs
```

(c) Code Generated

Figure 3. Algorithm for unknown loop bounds in Binaries

```
for i from lb_i to ub_i
  sum = 0;
  for j from lb_j to ub_j
   sum += sum + A[i,j]
  end for
end for
```

(a) Source code with scalar dependence

```
Loop_i : .............
  sum_r = 0;
  Loop_j : ..........
   tmp_r = *addr1
   sum_r += sum_r + tmp_r
  end for
end for
```

(b) Binary Code with scalar dependence

Figure 4. Scalar Dependence in code

scalar dependence on $Loop_j$, but not a scalar dependence on $Loop_i$. Hence $Loop_i$ can be parallelized in this code. Variable *sum* may be register allocated in a binary (say to sum_reg) and data flow will tell us that it is live across the $Loop_j$ but not live across $Loop_i$.

Some variables in the source code may be allocated to memory; these will be analyzed as memory references by theory presented in section II-B. They will likely appear as addresses with a constant base and no offset. The theory will handle their dependencies in a simple, degenerate case. However the dependencies get analyzed, every dependency that is present in a binary is analyzed and its effect on parallelization is accounted for.

V. Deciding Partitions

As we have shown in section II for the code in figure 1(a) and 1(c) the dependence vector is $\vec{D}$ = (1,0), indicating that there is a dependence in steps of 1 along i, whereas there is no dependence along induction variable j. So, if we execute all iterations of i on one processor then we can parallelize the iterations along j among all the processors. Pictorially, this is represented as Partition 1 in figure 5, which shows the iteration space as a 2-D matrix of i and j values. Conversely, in figure 1(b) the dependence vector is $\vec{D}$ = (0,2), indicating that there is a dependence in steps of two along induction variable j, and no dependence along induction variable i. So, if we execute all iterations of j on one processor then we can parallelize the iterations along i among all the processors. Pictorially, this is represented as Partition 2 in figure 5. Partition 3 in that figure can be used when there is no loop-carried dependence on either loop dimension (i.e. $\vec{D}$ = (0,0)). We also check that there is no register dependence at this dimension of the loop, as register dependencies prevent parallelism.

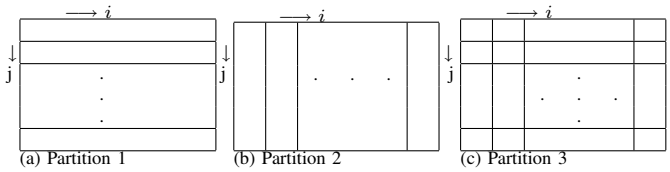

Figure 5. Different partitions of the iteration space

Improved performance is possible when loop transformations are included in the system, such as reduction, loop interchange, loop fusion, loop fission, strip mining, and loop skewing [6]. Once dependence vectors are computed, transformations can be applied much like from source code using any of the decision algorithms in the literature such as [6], which decide which transformations to apply and in what order. The choice of decision algorithm is orthogonal to this paper. For now we implement a simple decision algorithm for two transformations – reduction and strip mining – that we have found to work adequately for the benchmarks that we target. Implementing more transformations and a better decision algorithm can only improve our presented results further.

VI. Code Generation

After the the distance vectors are calculated, transformations done, and the loop dimension(s) to be parallelized are decided, code needs to be generated for each parallel loop dimension. Since the body of the loop is executed on all parallel threads, the most convenient and efficient code generation model is the Single Program Multiple Data (SPMD) model. The underlying idea is that the iterations of the loop are divided among threads; hence to keep the code-size increase to a minimum, the same code is executed on all threads using different loop bounds.

From source code, SPMD code can be generated by simply replacing the symbolic values of the lower and upper bounds of loop induction variables by new values. These methods are fairly straight forward as the symbolic information is readily available in source.

From binary code, code generation is conceptually similar to that from source. For each loop dimension to parallelize we calculate the new lower and upper bound using the formulae below.

$$\texttt{new\_lb}_{\texttt{addr\_reg}} = \texttt{Base} + \texttt{lb}_\texttt{i} * \texttt{size\_j} + \texttt{lb}_\texttt{j} * \texttt{elem\_size} + \frac{\text{PROC\_ID} * (\texttt{ub}_\texttt{j} - \texttt{lb}_\texttt{j}) * \texttt{elem\_size}}{\text{NPROC}} \quad (11)$$

$$\texttt{new\_ub}_{\texttt{addr\_reg}} = \texttt{min}\langle \texttt{ub\_j}, \texttt{new\_lb}_{\texttt{addr\_reg}} + \frac{(\texttt{ub}_\texttt{j} - \texttt{lb}_\texttt{j}) * \texttt{elem\_size}}{\text{NPROC}} \rangle \quad (12)$$

Replacing the bounds in (B) and (C) generates the parallel code to be executed on all NPROC processors. If the outer loop is partitioned, then statements (E) and (F) are similarly modified. Unlike loop partitioning, data partitioning is not necessary since we primarily target shared memory platforms common in multi-cores.

Generating parallel code requires the use of some parallel thread library. We implement POSIX-compliant *pthreads* calls, given that POSIX is a widely used portable industry standard, although any library can be used. POSIX-complaint parallel threads are created once at the start of *main()* in the binary, rather than at each loop to avoid paying the steep thread-creation cost multiple times. Only the main thread executes serial code between parallel loops. Parallel threads only execute loop code. When a parallel thread finishes one loop it waits for the main thread to inform it which loop to execute next in a broadcast. The broadcast also contains the values of registers calculated by the main thread that are needed by the parallel loop threads. A barrier is inserted into the binary at the end of every loop.

VII. Using LLVM for Implementation

Our binary rewriter translates the input x86 binary to the intermediate format of the LLVM Compiler [14], and then uses the x86 back-end LLVM to write the output binary. LLVM, which stands for Low-Level Virtual Machine, is a well-known, open-source compiler developed at the University of Illinois; it is now maintained by Apple Inc. *This conversion back to compiler IR is not a necessity for the work we present in this paper; any binary rewriter can use our theory*. However using LLVM IR enables us to use LLVM's rich infrastructure, such as control-flow analysis, dataflow analysis, and optimization passes, so that we did not have to write our own for the rewriter. Each instruction in the binary is converted to its equivalent LLVM IR instruction. The pushes and pops are analyzed to determine function arguments, caller and callee saves and stack accesses. Each stack frame present in the original binary is converted to an stack array in the intermediate IR. These techniques enable the addition of new stack variables in functions. The globals are accessed from their original addresses as we retain the original segments. Register allocated variables in the binary are converted to virtual registers.

The Loop Simplify and Loop Unswitch passes in LLVM are run to help the scalar evolution and induction variables on our code. Scalar Evolution and Induction Variable analysis in LLVM [14] help us in identifying the induction and derived induction variables in code. This helps us determine (A) to (F) required for distance vector calculation as described in section II. We also use the control flow and data flow information present in LLVM to our leverage to identify scalar dependencies, affine loops and shared/private variables for each loop.

An important side benefit of using LLVM is that it enabled us to do cross-ISA translation of code. We used LLVM's C backend to convert an input x86 binary to equivalent functional C code. This code (which is parallel using *pthreads* in our case) was then compiled using GCC on a 64-threaded SPARC T2 machine, and speedup was measured (see results section). The reason we did this was because SecondWrite presently implements only an X86 front-end. Of course this cross-ISA translation will not work in the general case when the code uses machine-specific library calls. However it worked for our programs since they only used only the platform-independent C and *pthreads* libraries.

To be clear, our LLVM's output C code generated from binaries is quite low-level and lacks array declarations and index expressions. Hence source parallelism methods will not work on it, necessitating our method.

VIII. Experimental Framework and Results

The input to our binary parallelizer are highly optimized (-O3) binaries compiled by GCC. These binaries don't contain any relocation or symbolic information. We have tested our parallelizer on benchmarks from *Polybench* (the Polyhedral Benchmark suite) and *Stream*(from the HPCC suite). We use three different machines to test our benchmarks. The machine descriptions are provided in table I. The benchmarks represent heavily used kernels in scientific and multi-media workloads.

Name	*CPUs*	*Cores/ CPU*	*Threads/ Core*	*Total Threads*	*Model*
DASH	1	4	2	8	Xeon E5530
BUZZ	4	6	1	24	Xeon E7450
T2	1	8	8	64	Ultra SPARC T2

Table I
Test machines

Our source parallelizer is implemented by feeding the parallelizer with the symbolic information present in a source. We then apply the same dependence analysis, partition techniques and code generation methodologies we have presented above. As the symbolic information is exploited to the fullest, we compare to state of the art affine parallelizers.

The speedups when parallelizing source and when parallelizing binaries with increasing number of threads on the different machines are presented in tables II, III, and IV; one figure per machine. Table II shows the speedup averages 5.7 when parallelizing from source code versus 5.1 when parallelizing from the x86 binary on the x86 DASH machine with 8 threads. This shows that (a) the speedups are nearly as effective from binaries as from source code, validating our theory; and (b) the speedups scale well. Table III shows the speedups from source and binary on 24 threads on the x86

BUZZ machine are 8.51 are 7.28 respectively. The speedups on BUZZ scale less well than DASH beyond 4 cores since the communication is out of the chip beyond 6 threads.

Table IV shows the speedups on a SPARC T2 machine average 15.37 when parallelizing source and 14.7 *when parallelizing an x86 binary and using our rewriter to convert it to a SPARC binary*. This cross-ISA-translation is done as described in section VII.

Further performance-related observations are as follows. On T2 beyond 8 threads the communication costs increase as we need to communicate between different cores. Also for some benchmarks (e.g. *gemver*) we observed that 64 threads run slower than 32 threads. The reasons for this could include the high communication costs between different cores or resource sharing (such as ALUs) between the 8 hardware-supported hyperthreads-like threads per core. We have collaborators are helping us collect system characteristics on different machines, which we will integrate in the future to do machine- and thread-specific optimizations; for example by not spawning more threads than the hardware can efficiently support for applications with certain characteristics.

Some benchmarks do not scale as well as others (such as *atax*) . The reason for this is that we parallelize the inner loop, and the resulting fine-grained threads for the comparatively small data set are not able to overcome barrier and broadcast latencies for the loop. In the future we plan to implement various loop transformations to improve these numbers further. The *covariance* benchmark parallelizes well from source but poorly from a binary. From source, the compiler can detect that different memory operations in the loop access different portions of the same array (upper triangle and lower triangle of a 2-D array), and hence are independent allowing parallelization. From a binary, the array's linearization prevents such discovery, so the accesses are conservatively deemed dependent. However we expect such cases to be very rare, and also with further linear algebraic techniques we will be able to correctly derive dependence vectors for most of these cases.

We present the statistical count of number of loops present in each benchmark and the number of loops parallelized successfully from source and binary in table V. The total number of loops counted are the outer loops present in benchmarks. Each outer loop may contain several nesting levels. Loops parallelized refers to one nesting level of the loop being parallelized.

IX. Related Work

This related work is short since it only lists potentially competing related work. Supporting related work is listed throughout the paper as appropriate. Here we discuss related work pertaining to (i) Binary rewriters and their applications. (ii)Dynamic Binary Automatic Parallelization Methods (iii) Affine based automatic parallelizing compilers from source

Benchmark		1	2	4	8	Benchmark		1	2	4	8
2mm	Source	1	1.85	3.75	4.78	bicg	Source	1	1.93	3.53	6.47
	Binary	0.98	1.76	3.59	4.85		Binary	1.1	1.63	2.99	5.73
atax	Source	1	1.42	2.01	3.54	doitgen	Source	1	1.8	3.56	7.65
	Binary	0.75	1.19	1.91	3.05		Binary	0.99	1.76	3.36	7.85
covariance	Source	1	1.29	2.3	3.98	gesummv	Source	1	1.67	2.82	6.86
	Binary	0.98	0.94	0.97	0.91		Binary	0.93	1.31	2.36	5.68
gemver	Source	1	1.73	3.14	7.53	correlation	Source	1	1.31	2.21	4.19
	Binary	0.9	1.64	3.03	7.25		Binary	0.96	1.31	2.1	3.77
jacobi-2d	Source	1	1.9	3.4	7.4	gemm	Source	1	1.84	3.67	5.6
	Binary	0.94	1.43	2.82	5.64		Binary	1.01	1.77	3.65	5.19
3mm	Source	1	1.8	3.64	5.01	stream	Source	1	1.89	3.61	5.91
	Binary	0.99	1.75	3.6	4.89		Binary	1.01	2	3.77	6.77
Average	Source	1	1.7	3.14	5.74	Speedup on DASH for 1 , 2 , 4 and 8 threads					
	Binary	0.96	1.54	2.85	5.13						

Table II
SPEEDUP ON X86 DASH FOR SOURCE AND BINARY

Benchmark		1	2	4	8	16	24	Benchmark	1	2	4	8	16	24
2mm	Source	1	1.97	3.83	6.32	11.79	15.6	bicg	1	2.1	3.55	4.59	4.31	3.97
	Binary	1	1.96	3.75	5.96	10.52	13.53		0.83	1.76	3.08	3.88	3.36	3.01
atax	Source	1	1.57	2.42	3.27	3.5	3.09	doitgen	1	1.99	3.98	6.58	12.59	15.7
	Binary	0.75	1.01	1.72	2.47	2.9	2.73		0.99	1.98	3.95	6.6	12.53	16.12
covariance	Source	1	1.31	2.22	4.08	7.49	10.05	gesummv	1	1.54	2.35	3.38	2.87	2.73
	Binary	1	1.89	3.42	4.85	4.23	2.8		0.74	1.11	1.8	2.49	2.89	2.18
gemver	Source	1	1.48	2.56	3.81	4.18	4.08	correlation	1	1.32	2.24	3.96	7.17	9.86
	Binary	0.93	1.43	2.42	3.8	4.34	4.02		1	1.29	2.12	3.51	5.74	7.46
jacobi-2d	Source	1	1.89	3.29	5.02	5.01	4.56	gemm	1	1.97	3.89	6.59	11.29	15.67
	Binary	0.63	1.24	2.16	3.61	4	4.02		1	1.95	3.8	6.28	11.52	15.11
3mm	Source	1	1.97	3.82	6.34	11.85	15.82	stream	1	2.29	3.14	3.47	1.87	0.98
	Binary	1	1.98	3.83	6.35	11.6	15.23		0.95	1.99	2.97	3.12	2.02	1.16
Average	Source	1	1.78	3.11	4.78	6.99	8.51	Speedup on BUZZ for 1 , 2 , 4 , 8 , 16 and 24 threads						
	Binary	0.9	1.62	2.92	4.41	6.3	7.28							

Table III
SPEEDUP ON X86 BUZZ FOR SOURCE AND BINARY

Benchmark		1	2	4	8	16	32	64	Benchmark	1	2	4	8	16	32	64
2mm	Source	1	1.99	3.99	7.96	15.21	26.31	36.95	bicg	1	1.95	3.88	6.7	9.44	5.01	1.62
	Binary	1.03	2.05	4.09	8.14	15.65	27.44	39.62		1.04	2.08	4.07	7.3	9.28	7.89	1.15
atax	Source	1	1.4	1.74	2	1.97	1.65	0.83	doitgen	1	2	3.99	7.96	15.66	26.57	24.2
	Binary	1.04	1.44	1.79	2.01	1.85	1.85	0.86		1.03	2.05	4.09	8.13	16.2	28.01	21.05
covariance	Source	1	1.34	2.29	4.26	8.12	15.3	23.25	gesummv	1	1.97	3.91	7.56	11.69	10.06	3.26
	Binary	1	1	1	1	0.99	0.93	0.81		0.99	1.96	3.83	7.49	11.14	10.03	3.63
gemver	Source	1	1.99	3.92	7.42	11.97	14.84	9.65	stream	1	1.95	3.83	7.03	10.77	9.77	2.29
	Binary	0.99	1.97	3.74	7.12	12.1	15.53	7.63		0.98	1.96	3.84	7.04	10.89	9.86	1.52
jacobi-2d	Source	1	1.83	3.46	5.82	7.81	8.63	2.65	correlation	1	1.34	2.29	4.24	8	14.88	23.62
	Binary	0.91	1.79	3.1	5.6	8.07	8.12	2.34		0.9	1.23	2.12	3.94	7.37	13.15	20.98
3mm	Source	1	2	4	7.95	15.2	26.45	40.58	gemm	1	1.98	3.97	7.93	15.37	24.99	33.9
	Binary	1.03	2.05	4.09	8.16	15.63	27.2	40.22		0.99	1.99	3.96	7.9	14.98	26.41	35.58
Average	Source	1	1.81	3.44	6.4	10.93	15.37	16.9	Speedup on T2 for 1 , 2 , 4, 8 , 16 , 32 and 64 threads							
	Binary	0.99	1.8	3.31	6.15	10.35	14.7	14.62								

Table IV
SPEEDUP ON SPARC T2 FOR SOURCE AND X86 BINARY

Benchmark	Total Number of Loops	Number of Loops parallelized from source	Number of Loops parallelized from binary	Benchmark	Total Number of Loops	Number of Loops parallelized from source	Number of Loops parallelized from binary
2mm	7	7	7	bicg	3	3	3
atax	3	3	3	doitgen	3	3	3
covariance	4	4	3	gesummv	2	2	2
gemver	5	5	5	correlation	5	5	4
jacobi-2d	2	2	2	gemm	4	4	4
3mm	10	10	10	stream	3	3	3

Table V
ANALYSIS OF LOOPS PRESENT IN THE BENCHMARKS

and (iv) Techniques to calculate distance and direction vectors.

Binary rewriters Existing binary and object-code rewriters include Etch [15], squeeze and squeeze++ [16], [17], PLTO [18], DIABLO [19], ALTO [20] and spike [21], [22]. These infrastructures have been used for a variety of code improvements including code size reduction, performance improvement and security enhancements.

Dynamic Binary Automatic Parallelization Methods Existing Binary Automatic Parallelization techniques are limited to dynamic methods and do not perform sophisticated affine analysis like we do. Yardimci and Franz [7] present a method to dynamically compile a sequential binary to a parallelized or vectorized code. Their techniques are mostly complementary to ours, in that instead of affine parallelism, their techniques include control speculation, loop distribution and automatic parallelization of recursive techniques. They parallelize loops that do not have loop carried dependencies, which limits the scope of loops parallelized drastically. As, we are able to perform sophisticated affine analysis on memory strides present in loops, the scope of loops parallelized by us is higher. Further, their techniques are dynamic preventing them from integrating sophisticated decision algorithm into their system.

Affine based Automatic Parallelizers Affine-based parallelizing compilers such as Polaris [23], SUIF [24], [25], [26], and pHPF [27], PROMIS [28] and Parafrase-2 [29] have been built in the past. All these automatic parallelizing compilers parallelize code from source, unlike our method. As acknowledged throughout this paper, our method builds on existing methods, but has significant differences allowing it to work on binaries for the first time.

Distance and Direction Vector Calculation Affine loop parallelism has required solving systems of linear diophantine equations [12] to calculate distance vectors. Various techniques have been proposed in literature to solve these equations. These include the Greatest Common Divisor (GCD) test [11], [8] , Banerjee's inequalities [8], Single Index and Multiple Index Tests [12], [9], Multidimensional GCD [8], the delta test [13] and the omega test [30]. We adopt these tests from the source to our binary automatic parallelizer. We have presently implemented the Greatest Common Denominator, Single and Multiple Index tests to solve the linear diophantine equations that we recover directly from a binary.

X. CONCLUSIONS AND FUTURE WORK

We have taken binary programs without relocation or symbolic information and parallelized them by recognizing affine loops. *This work to our knowledge is the first of its kind, as no past work has performed affine analysis directly from binaries.* We take ideas from source affine loop parallelism but adapt them to binaries. This is a significant step to understanding all the information present in compiled binaries and leveraging it to parallelizing them.

In the future we intend to add many more transformations, such as loop fusion, loop fission, loop interchange, loop reversal, and loop skewing to our parallelizer. We do not foresee any theoretical issues in adopting these techniques to binaries, but due to lack of time we have not implemented them for this paper. We have implemented the recognition of commutative reduction from binaries and the results presented includes it.

In future, we also intend to apply graph-based parallelizers to binaries, to extend the scope beyond affine loops. We will adapt the work on graph-based parallelism [31] to a binary rewriter since ours are limited to scientific programs and theirs are not.

The other scope of research that we are focusing on is platform specific parallelizing. Our collaborators help us collect system characteristics such as effective number of contexts, best barriers, cache characteristics etc that we include in our parallelizer. We already have techniques to use the best barrier and effective number of contexts for different platforms. We view this work as opening up a new direction of research that will enable further optimizations directly on a binary.

ACKNOWLEDGMENT

This material is based upon work partially supported by the United States Air Force under Contract No. FA8650-09-C-7918, and in part by NSF CNS grants #0720683 and #0916903.This work is protected by US Patent pending #12/771,460 titled "Automatic Parallelization using Binary Rewriting", filed April 30, 2010. It may not be reproduced for any purpose without explicit permission from the University of Maryland.

REFERENCES

[1] Iyle, "Cpu trends," vol. Tech talk, May 13, 2008, http://techtalk.pcpitstop.com/research-charts-cpu/.

[2] O. A. R. Board, "OpenMP C and C++ application program interface, version 1.0," http://www.openmp.org, 1998.

[3] O. R. Board, "OpenMP Fortran application program interface, version 2.0," http://www.openmp.org, 2000.

[4] "The Message Passing Interface (MPI) standard," vol. Consortium website, 2007, http://www-unix.mcs.anl.gov/mpi/.

[5] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili, "Parallel programmer productivity: A case study of novice parallel programmers," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 35.

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

[7] 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.

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

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

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

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

[12] U. Banerjee, "Speedup of ordinary programs," Ph.D. dissertation, Champaign, IL, USA, 1979.

[13] 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.

[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (GCO)*, 2004, pp. 75–87.

[15] T. Romer, G. Voelker, D. Lee, A. Wolmen, W. Wong, H. Levy, and B. N. Bershad, "Instrumentation and optimization of win32/intel executables," *USENIX Windows NT Workshop*, August 1997.

[16] B. D. Sutter, B. D. Bus, and K. D. Bosschere, "Link-time binary rewriting techniques for program compaction," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 5, pp. 882–945, 2005.

[17] S. Debray, W. Evans, and R. Muth, "Compiler techniques for code compression," University of Arizona, Tucson, AZ 85721, USA, Tech. Rep., April 1999, http://citeseer.ist.psu.edu/243038.html.

[18] B. Schwarz, S. Debray, and G. Andrews, "Plto: A link-time optimizer for the intel ia-32 architecture," *In Proc. 2001. Workshop on Binary Rewriting (WBT)*, 2001, citeseer.ist.psu.edu/schwarz01plto.html.

[19] V. P. L, C. D, D. S. B. De Bus B, and D. B. K, "Diablo: a reliable , retargetable and extensible link-time rewriting framework," in *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005*. IEEE, 2005, pp. 7–12.

[20] R. Muth, S. K. Debray, S. Watterson, and K. D. Bosschere, "Alto: a link-time optimizer for the compaq alpha," *Softw. Pract. Exper.*, vol. 31, no. 1, pp. 67–101, 2001.

[21] R. C. et al., "Spike: an optimizer for alpha/nt executables," in *NT'97: Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*. Berkeley, CA, USA: USENIX Association, 1997, pp. 3–3.

[22] R. S. Cohn, D. W. Goodwin, and P. G. Lowney, "Optimizing alpha executables on windows nt with spike," *Digital Tech. J.*, vol. 9, no. 4, pp. 3–20, 1998.

[23] 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.

[24] 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.

[25] M. W. H. et al., "Maximizing multiprocessor performance with the suif compiler," *Computer*, vol. 29, no. 12, pp. 84–89, 1996.

[26] M. H. H. et al., "Detecting coarse-grain parallelism using an interprocedural parallelizing compiler," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM Press, 1995, p. 49.

[27] 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 *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM Press, 1995, p. 71.

[28] H. S. et al., "The design of the promis compiler," in *CC '99: Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*. London, UK: Springer-Verlag, 1999, pp. 214–228.

[29] C. D. P. et al., "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.

[30] 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.

[31] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August, "Revisiting the sequential programming model for the multicore era," *IEEE Micro*, vol. 28, no. 1, pp. 12–20, 2008.