Polyhedral Parallelization of Binary Code

BENOIT PRADELLE, ALAIN KETTERLIN, and PHILIPPE CLAUSS,
INRIA Nancy Grand Est and LSIIT, Université de Strasbourg, France.

Many automatic software parallelization systems have been proposed in the past decades, but most of them are dedicated to source-to-source transformations. This paper shows that parallelizing executable programs is feasible, even if they require complex transformations, and in effect decouples parallelization from compilation, for example, for closed-source or legacy software, where binary code is the only available representation.

We propose an automatic parallelizer, which is able to perform advanced parallelization on binary code. It first parses the binary code and extracts high-level information. From this information, a C program is generated. This program captures only a subset of the program semantics, namely, loops and memory accesses. This C program is then parallelized using existing, state-of-the-art parallelizers, including advanced polyhedral parallelizers. The original program semantics is then re-injected, and the transformed parallel loop nests are recompiled by a standard C compiler.

We show on the PolyBench benchmark suite that our system successfully detects and parallelizes almost all the loop nests from the binary code, using a recent polyhedral loop parallelizer as a backend. The paper ends by elaborating a strategy to parallelize more complex programs, such as those containing non-linear accesses to memory, and provides a few example case-studies.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*Optimizations*

General Terms: Algorithms, Performance

ACM Reference Format:

Pradelle, B., Ketterlin, A., and Clauss, P. 2012. Polyhedral parallelization of binary code. *ACM Trans. Architect. Code Optim.* 8, 4, Article 39 (January 2012), 21 pages.
DOI = 10.1145/2086696.2086718 <http://doi.acm.org/10.1145/2086696.2086718>

1. INTRODUCTION

Shared memory multi-processors are now ubiquitous, and there is a strong pressure on software developers to write programs that are able to use as much of the available computing power as possible. However, writing parallel programs is hard and existing automatic parallelization techniques place strong constraints on programs they handle and/or on the programming environment they use.

Most of the studies on automatic parallelization have focused on source-to-source transformation tools. Sophisticated techniques have been developed for some classes of programs, and rapid progress is made in the field. However, there is a persistent hiatus between software vendors having to distribute generic programs, and end users running them on a variety of hardware platforms, with varying levels of hardware parallelism available. The next decade may well see an increasing variety of parallel hardware, as it has already started to appear in the embedded systems market. At the same time, one can expect more and more architecture-specific automatic

Authors' address: B. Pradelle, A. Ketterlin, and P. Clauss, CAMUS Group, INRIA Nancy Grand Est and LSIIT, Université de Strasbourg, France; email: [pradelle, ketterlin, clauss]@icps.u-strasbg.fr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1544-3566/2012/01-ART39 \$10.00

DOI 10.1145/2086696.2086718 <http://doi.acm.org/10.1145/2086696.2086718>

Fig. 1. (1) Analyzable portions of the sequential binary program are raised to simple C loop nests only containing the correct memory accesses, and (2) sent to a back-end source-to-source parallelizer. (3) The resulting parallel nest is filled with actual computations on memory, and compiled using a standard compiler.

parallelization techniques, e.g., GPU or FPGA oriented scheduling algorithms. Therefore, the widening gap between software production and execution becomes a real problem in the adoption of parallelism as a means to efficiently build, deploy, and use computing infrastructures.

This paper uses an approach that could be named parallelization as a service. In this setting, the operating system or runtime environment provides tools that take sequential binary programs and transforms them into parallel executable code. The transformation is performed statically, e.g., at installation time, and may use any source-to-source parallelization back-end. As it is performed on binary programs, closed-source third party applications, legacy programs, and libraries used by the program can be parallelized, independently from their original source language.

The strategy adopted by our system can be summed up in three phases, depicted in Figure 1. The *raising phase* brings the binary code into a C program where only the loop nests and the memory references are present. The *parallelizing phase* takes that result and generates a set of transformations to apply to the code. Those transformations are decided by a standard source-to-source parallelizer such as CETUS [Bae et al. 2009], a classical loop nest parallelizer, or PLUTO [Bondhugula et al. 2008], a parallelizer using the polyhedral model. Finally, the *lowering phase* is in charge of converting the resulting parallel nests to a program where the original operations on memory are effectively performed. This final parallel C program is then compiled using a standard C compiler such as GCC.

The main contributions of this paper are many fold. First, our parallelization system is purely static, contrary to a majority of existing binary code parallelizers, which are dynamic speculative systems. Thus, our implementation does not suffer from any runtime overhead and does not require any specific hardware, even though it can nicely complement the existing dynamic systems. Second, our system can exploit any existing source-to-source back-end parallelizer, including polyhedral parallelizers, leading to the first attempt on advanced loop transformations on binary programs. Third, we propose some techniques to extend binary program parallelization towards complex codes where the memory references can be arbitrarily complex parametric polynomials. Fourth, we evaluate an implementation, taking as an input a x86-64 binary program, analyzing it with advanced methods, parallelizing it with one of several back-end source-to-source parallelizers, and re-compiling the parallel code back in the binary program, using a standard C compiler.

Section 2 explains how the binary code is analyzed, and what kind of intermediate representation is extracted. Since the target is mostly on polyhedral transformations, the paper focuses on recovering affine loop nests. Section 3 details the parallelizing phase, using an external parallelizer. Because we wanted the whole system to remain modular, this phase includes a set of adaptation passes that convert the intermediate representation into a form acceptable to our example back-end parallelizers. The lowering phase is also described in this section. Detailed experiments are described in Section 4. Because this study is almost entirely targeting compute-intensive kernels and polyhedral parallelization techniques, we devote Section 5 to a discussion on how

the same binary parallelization strategy could be applied to other classes of programs. We suggest several extensions, and use two programs from a SPEC suite to illustrate them.

2. DECOMPILING X86-64 EXECUTABLES

To be able to automatically parallelize a program, one must perform dependence analysis, and to be able to perform dependence analysis one must have a compact representation of the memory addresses the program accesses. A large majority of static parallelizers use multi-dimensional linear functions of parameters and loop counters as the basic access function model. Therefore we need to first locate loops in the code, and then extract access functions. This section details this process of memory behavior extraction.

The first step handles one routine at a time, extracting basic block boundaries and a complete control-flow graph (CFG). A dominator tree is computed from the CFG, and natural loops are recognized and organized into a hierarchy. Routines where the CFG cannot be reconstructed are discarded and will not be transformed. Similar passes appear in virtually every optimizing compiler and excellent descriptions of their details are available [Appel and Ginsburg 2004; Muchnick 1997].

The next step is a data-flow analysis, where each instruction is analyzed to extract the sets of variables it uses and defines. When dealing with binary code, all machine registers (including flags) are considered, and a single monolithic variable called M is used to represent memory. Each definition of M (i.e., each memory store) implicitly uses the currently visible definition of M , i.e., M is weakly updated. The Static Single Assignment (SSA) form of the program is then computed, using essentially the original algorithm [Cytron et al. 1991]. The result is a set of use-def links, as well as new ϕ -functions placed on appropriate basic blocks entry point. The SSA form is the basic intermediate representation on which the rest of the processing takes place.

2.1. Memory Access Description

Because the goal of the decompilation phase is to obtain a symbolic description of all memory addresses used by the program, the first step is to understand how registers involved in addressing are combined. Use-def links of the SSA form provide a direct way to determine how any address used in the program is computed. On x86-64 architectures, addresses appearing in the code are of the form $Base + s \times Index + o$, where $Base$ and $Index$ are register names and s and o are small immediate values (all terms are optional). Starting with this expression, one can recursively follow use-def links from the uses of $Base$ and $Index$ to rebuild a symbolic expression using definitions that dominate the memory access. For instance, an instruction like `ADD r12, rbx` defines $r12^{42}$ as the expression $r12_{41} + rbx_7$ (superscripts indicate definitions and subscripts indicate uses). This recursive substitution process is applied to all memory operands in all instructions, effectively traversing all instructions that participate in address computation (and ignoring others).

Given a representation model, e.g., the integer linear model, this recursive substitution stops in four distinct situations:

- (1) when reaching the routine entry point, i.e., when using an input parameter;
- (2) when reaching an instruction that, after parsing, would lead to an expression that is outside the representation model, e.g., a bit-masking, or floating point instruction;
- (3) when reaching one of the ϕ -functions introduced by the SSA form.
- (4) when reaching a definition that uses memory;

The first two situations are strict limitations: The first is due to the fact that we use an intra-procedural analysis, and the second is due to the expressiveness of our description

model (note that this limitation applies only to instructions that are actually involved in address computations). The last two limitations, however, can be overcome. The next section explains how induction variable resolution, or *scalar evolution*, replaces ϕ definitions with expressions involving loop counters. Later sections describe techniques to solve cases where address computations involve memory cells.

2.2. Induction Variable Resolution

This section explains how the third limitation to address expression reconstruction, namely ϕ -functions, can be overcome. A ϕ -function at the head of a loop often expresses the fact that the register (or memory cell) “enters” the loop with an initial value, which is then modified at each iteration. Because we target regular programs manipulating arrays, ϕ -functions used in address computations are, more often than not, linear induction variables. Consider the following typical simple example, where the line starting with # is a comment giving the definition of the ϕ -function:

```

    mov r94, 0x20
L:
    # r95 =  $\phi(r9_4, r9_6)$ 
    ...
    add r96, 0x8
    jmp L

```

A closed form for $r9^5$ is easy to derive: starting with value 0x20 and incremented by 0x8 at each iteration, $r9^5$ has value $0x20 + i \times 0x8$ at the i^{th} iteration.

Therefore, to solve induction variables, we introduce a new, unique virtual counter for each loop, and consider all ϕ -functions used in address computations. For any ϕ -function $r = \phi(r_1, r_2)$, where r_2 is the value defined inside the body of the loop:

- if r_2 is of the form $r + \alpha$ (after complete expression substitution) with α being a loop-invariant quantity, the definition of r becomes $r_1 + i \times \alpha$;
- if r_2 is of the form $\beta + i \times \gamma$, with both β and γ loop-invariant quantities, this expression becomes the definition of r if r_1 and β are identical.

In both cases, i is the normalized loop counter, with initial value zero and incremented by one at each new iteration. Once the definition of a ϕ -function is known, every occurrence of the ϕ -function inside the loop body can be replaced by its definition, thereby replacing loop-varying register occurrences with the loop counter and loop-invariant expressions.

Figure 2 shows an extract from the `swim.m` benchmark of the SPEC OMP-2001 benchmark suite. Comments in the code show how memory access descriptions are formed, and how induction variables are solved. After register substitution and induction variable resolution, all memory accesses are expressed in terms of normalized loop counters, along with register definitions that cannot be further substituted. However, in some cases, these definitions use a value stored in some memory cell, which often is a stack slot. The next section explains how to deal to such cases.

2.3. Tracking Stack Slots

All but the simplest programs use memory to store intermediate results, which are later used in address computations. These memory cells are almost always stack slots. A typical example is given in the following code fragment (comments give the address expressions obtained so far, where `rsp.1` is the value of the stack pointer upon routine entry, and `I` is some loop counter).

```

mov r15, 0x1 # r15.2 = 0x1
r15.3 =  $\phi(r15.2, r15.4)$  #  $\rightarrow (b) 1 + I$ 
[...]
mov rsi, 0x1 # rsi.3 = 0x1
rsi.4 =  $\phi(rsi.3, rsi.5)$  #  $\rightarrow (b) 1 + J$ 
lea r11, [rsi+0x1] # r11.4  $\rightarrow (a) rsi.4 + 0x1$ 
[...]
movsd xmm0, q[rax+r9*8]
#  $\rightarrow (a) 0xe28d4b0 + 8*rsi.4 + 30416*r15.3$ 
#  $(b) 0xe294b88 + 8*J + 30416*I$ 
mulsd xmm0, xmm4
mulsd xmm0, q[rax+rdx*8]
#  $\rightarrow (a) 0x5fba70 + 8*rsi.4 + 30416*r15.3$ 
#  $(b) 0x603148 + 8*J + 30416*I$ 
movsd q[rax+rdx*8+0x...], xmm0
#  $\rightarrow (a) 0x3e68b090 + 8*rsi.4 + 30416*r15.3$ 
#  $(b) 0x3e692768 + 8*J + 30416*I$ 
[...]
mov rsi, r11 # rsi.5 = r11.4  $\rightarrow (a) rsi.4 + 0x1$ 
[...]
add r15, 0x1 # r15.4  $\rightarrow (a) r15.3 + 0x1$ 

```

Fig. 2. Symbolic address expressions in a double loop in binary code in SSA form: (a) after recursive register substitution, and (b) after induction variable resolution. Indentation is used to highlight the inner loop. Comments in the code provide some intermediate results (after #).

```

1. mov [rsp+0x20], rax # address = rsp.1 - 0x38
...
2. mov [rcx+8*r9], xmm1 # address = rsi.1 + 30416*I
...
3. mov r12, [rsp+0x20] # address = rsp.1 - 0x38
...
4. movsd xmm0, [r12]

```

In this fragment, register r12 is used to address memory at line 4. Recursive substitution of r12 is not possible because of the use of a memory operand ([rsp+0x20]). Even though it seems obvious on this fragment that the value assigned to r12 is the value of rax used in the instruction at line 1, the substitution process must be able to infer that the instruction at line 2 does not interfere with values stored at rsp+0x20: in x86-64 code, no implicit rule can ensure this property on the base of the fragment shown above. It has thus to be proved by analysis.

To have a tractable and fast memory cell tracker, we distinguish two separate regions: (1) the current stack frame, which is expected to hold intermediate address computations that we would like to track; and (2) the rest of the address space, which is expected to hold program data. Our analysis proceeds in two steps: first find out which register points to which region, at all program points, and second use this information to find the “last write” to any memory cell used at some point of time in an address computation.

The first problem can be solved by associating two bits to each register definition, indicating whether the register may contain an address inside the corresponding region. At routine entry, the rsp register is known to contain an address to the current stack frame, and all other registers are supposed to point to the rest of the address space: we assume the program conforms to the ABI in this respect. The “points-to” information is then propagated along all control-flow edges until a fixed point is reached. For each instruction and ϕ -function, each bit of the defined registers receive the logical OR of

the corresponding bits of all used registers. Memory has two pairs of bits, one for the stack frame, the other for the rest of the address space. In practice, this simple analysis is remarkably robust and surprisingly precise.

Typically, in the example given above, the result would be that both occurrences of `rsp.1-0x38` represent addresses located in the current stack frame (because so does `rsp.1`), and that address `rsi.1 + 30416*I` represents an address *not* pointing to the current stack frame. This is enough to assume independence between the use of memory at line 3 and the definition of memory at line 2.

Once we know where every register points to, the source of the value of a memory cell can often be found, especially if that cell is in the current stack frame. Starting with a use like `[rsp+0x20]` in the instruction at line 3 above, all that is needed is to be able to follow the chains of the memory stores, from nearest to farthest, checking whether we have found the specific cell we are looking for. We use a simple decision procedure: If both addresses point to distinct regions, they cannot correspond; otherwise, if the difference of their address expressions is non constant, we cannot decide and take a conservative “may” decision; otherwise, if the difference is zero we have found the last write, and if the difference is non zero we can continue searching at the previous memory store. This simple dependence test is clearly tailored to locate stack slots, whose addresses are usually of the form `rsp+ $\alpha$` , and will give poor results on other memory cells. That’s exactly what it was designed to achieve, and gives excellent results on the programs we have tested. In our example, addresses at lines 1 and 3 both resolve to the same expression, namely `rsp.1-0x38`, and we can insert a synthetic use-def link between both memory operands.

Once stack slots have been located, they behave as “virtual registers”. In particular, new ϕ -functions are introduced when appropriate, and induction variable resolution treats them exactly as regular register ϕ -functions.

2.4. Branch Conditions and Block Constraints

The last aspect of the binary code that needs to be captured are the conditions that govern the control-flow inside loop bodies, from which we will derive loop bounds. We need to distinguish two notions.

- A *branch condition* is a simple logical comparison between zero and an expression, with one of the operators `<`, `≤`, `>`, `≥`, `=` and `≠`; branch conditions are typically parsed from conditional jump instructions.
- A *block constraint* is a logical combination of branch conditions, of arbitrary complexity, typically represented in disjunctive normal form.

Branch conditions are directly extracted from the binary. Consider the following fragment of code appearing inside a loop with normalized counter `I` (the comment shows the result of induction variable resolution on a stack slot):

```
L:  # M[rsi+0x8] = 0x0 + 0x8*I
    ...
    cmp r152, [rsi+0x8]
    jg L
```

From this fragment, the branch condition is `0x8*I < r152`.

Block constraints are constructed with the help of control-dependence, which is given by the postdominator tree. Control dependence has an interesting property for our purpose: any edge $u \rightarrow v$ such that v is not the immediate postdominator of u (i.e., edges that move “out of a conditional jump”) imply a control dependence on all blocks that postdominate v without also postdominating u [Muchnick 1997, ch. 9]. In plain words, it means that all these blocks are executed only when the condition of going from u to v is true, provided the control has reached u . If we note $C[u]$ the constraint applying

to block u , and $\langle u \rightarrow v \rangle$ the condition of going from u to v , then control dependence infers that the constraint $C[u] \wedge \langle u \rightarrow v \rangle$ has to be placed on the blocks in question.

To compute constraints on all blocks of a loop with head h , we first set $C[h]$ to true, and $C[b]$ to false for all $b \neq h$. We then consider the loop body, i.e., the sub-graph of the CFG restricted to the blocks of the loop with back-edges removed: this is an acyclic graph, which therefore defines a topological order on the blocks. The blocks are then traversed in that order. For each block ending with a conditional jump, we apply the constraint propagation just described. After all blocks have been processed, constraints will have been propagated to all blocks of the loop. The constraints propagated back to the head of the loop will define the condition under which a new iteration is started: if this condition contains only the loop counter and loop-invariant quantities, it defines the loop trip-count.

For many common cases, the procedure outlined above produces an expression for the loop bound. Edges branching out of the loop are often placed in such a way that the whole loop can be translated into a for-loop, i.e., the conditions on the loop counter are the same for all blocks containing “useful” instructions (instructions that do something else than computing addresses). Conditional control-flow inside loops is preserved. The whole decompilation procedure has proved to be extremely effective for the programs we target. The next section has several examples of C code that is directly derived from the SSA intermediate representation, and that can serve as input to the next phase.

3. PARALLELIZING TRANSFORMATIONS

3.1. The Polyhedral Model

Among all the loop nests present in programs, some of them are said to be *affine*. They are made of loops whose bounds and memory access functions are affine expressions of global parameters and outer loop iterators. In the *polyhedral model*, those loop nests can be represented as polyhedra whose facets are defined by the loop bounds. Those polyhedra are called *iteration domains*. For example, consider the following loop nest:

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    ...
```

In this example, the corresponding polyhedron is a $N \times M$ rectangle. Each point with integer coordinates inside the rectangle represents an iteration of this loop nest. For instance, the point at coordinates (1, 5) represents the loop nest iteration when $i = 1$ and $j = 5$.

In this representation, executing a loop nest consists in scanning the integer points in the lexicographic order of the loop indices. A semantically equivalent loop nest can be generated by changing the scanning order, or *schedule*, in a way where all the inter-iterations dependencies appearing in the original loop nest are respected. These dependencies can automatically and exactly be computed. Polyhedral compilers such as PLUTO are able to determine the dependences and build a valid schedule in which the loop nest can be parallelized. This new schedule is built as a combination of loop reversal, interchange, and skewing of the original loop nest. More details are provided for instance in Feautrier [1992].

Although this model cannot be applied on any loop nest, it enables advanced optimizations on those which are handled, such as optimizing data locality, or transforming the loops to exhibit parallelism. There are several available tools to extract information from such nests or to transform and parallelize them. Our system intensively uses this model and some of its related tools when the loop nest is affine. We specifically use ISL [Verdoolaege et al. 2007; Verdoolaege 2010] as a toolbox to perform most of the related computations and PLUTO [Bondhugula et al. 2008] is one of our back-end

```

for (t1 = 0; -511 + t1 <= 0; t1++)
  for (t2 = 0; -511 + t2 <= 0; t2++) {
    M[10494144+4096*t1+8*t2] = 0;
    xmm1 = 0;
    for (t3 = 0; -511 + t3 <= 0; t3++) {
      xmm0 = M[6299808+4096*t1+8*t3];
      xmm0 = xmm0  $\odot$  M[8396960+8*t2+4096*t3];
      xmm1 = xmm1  $\odot$  xmm0;
    }
    M[10494144+4096*t1+8*t2] = xmm1;
  }

```

Fig. 3. Matrix multiply as it is extracted from the binary code.

parallelizer. PLUTO is a polyhedral source-to-source compiler, able to automatically compute a valid code transformation that exhibits parallel loop levels and optimizes data locality at the same time. It takes as input a sequential C code where target loop nests have been marked, and outputs a transformed C code parallelized using OpenMP directives [openmp 2008]. The other back-end parallelizer, CETUS 1.3 [Bae et al. 2009], does not use the polyhedral representation, but rather some classical approximate dependence tests to parallelize the loop nests without transforming them. Many other parallelizers could have been considered, but CETUS and PLUTO are typical examples, as they use very different approaches to parallelize loop nests.

3.2. Tailoring the Intermediate Representation

The analysis steps provide enough information to build a C code equivalent to the original binary representation of loop nests: for-loops can be reconstructed and access functions to memory are known. We can then rebuild a C code for each targeted loop nest and use existing source-to-source parallelizers to parallelize them. However, the current polyhedral tools put some restrictions on their input C code that force us to refactor the code we generate from the binary program.

First, the memory accesses extracted by the analysis steps all refer to a unique one dimensional array M whose base address is zero. As the programs' data segments are usually far from address zero, huge values can appear in the memory access functions. This often makes the dependence analyzers fail. We alleviate this issue in the following way. When the loop bounds are numeric constants, we can determine the address ranges of each memory reference. Thus, we can split the unique memory array M into several different arrays defined over non-overlapping chunks of memory.

Second, the C code reconstructed from the binary program does not contain the operations on data. Indeed the processor instructions used in the binary code have often no equivalent representation in pure C code. This is not a problem anyway as, most of the time, the parallelizers only exploit the data and control flows information. To create a parsable C program, we consider that every operation is equivalent to a neutral operation, $\odot$. This operation can be implemented by any operator in the actual C code sent to the parallelizer, for example "+". Each instruction in the binary code is then represented as an equality where the registers are C variables, the memory accesses are array accesses. The written (defined) operand is on the left side of the equality, while the read (used) ones are on the right side, combined by $\odot$.

We show in Figure 3, the C code extracted from a matrix multiply binary program. In the next figures, the memory array M is split into three disjoint arrays, and the operations on data are replaced by the neutral operator $\odot$.

Third, every write to a register usually cause data dependences, while this scalar reference can often be privatized. Although privatization is a simple task in


```

    for (t1 = 0; -511 + t1 <= 0; t1++)
      for (t2 = 0; -511 + t2 <= 0; t2++) {
        A2[4096*t1+8*t2] = 0;
        xmm1 = 0;
        for (t3 = 0; -511 + t3 <= 0; t3++)
          xmm1 = xmm1  $\odot$  (A1[4096*t1+8*t3]  $\odot$  A3[8*t2+4096*t3]);
        A2[4096*t1+8*t2] = xmm1;
      }

```

Fig. 4. Matrix multiply after simplifying the memory accesses and forward substitution.

ALGORITHM 1: Scalar to array replacement algorithm

```

foreach scalar  $s$  whose first access is a write do
   $fsref \leftarrow$  first reference to  $s$ ;
   $lsref \leftarrow$  last reference to  $s$ ;
  foreach memory write  $mw$  occurring after  $lsref$  do
     $dependence \leftarrow$  false;
    foreach memory access  $ma$  occurring after  $fsref$  and before  $mw$  do
      if  $mw$  intersects with  $ma$  then
         $dependence \leftarrow$  true;
      end
    end
    if not  $dependence$  then
      replace  $s$  by the element written by  $mw$ ;
      continue with the next  $s$ ;
    end
  end
end

```

non-transforming parallelizers, it is much more complex in the case of polyhedral compilers, because statements may be moved around by the scheduling transformation. At the time we performed our experiments, its support appeared to be broken in two of the most advanced parallelizing compilers: PLUTO and PoCC [Pouchet et al. 2010]. Thus, we remove as many scalar references as possible before transmitting the C code to the back-end parallelizer.

The first scalar optimization applied is induction variable resolution. It allows us to express a register value as an affine function of outer loop indices and other registers. All those registers are then replaced by their equivalent affine function in the C code and the instruction defining the register value is ignored.

Forward substitution is also applied, helping to remove many scalar dependences. Figure 4 shows the matrix multiply code after forward substitution. Notice that references to scalar `xmm0` have been removed. However, the `xmm1` variable is still present, provoking data dependences at every iteration.

The last scalar simplification tries to replace the remaining scalar references by array references, usually generating less data dependencies. We consider privatizable scalars, whose first reference in the loop nest is a write, unused after the loop nest. For each of such scalars, we look for a memory element always overwritten after the last reference to this scalar, but which does not alias the other memory elements accessed during the scalar life-span. If such memory element exists, it can be used in place of the scalar. The detailed algorithm is given in Algorithm 1.

```

for (t1 = 0; -511 + t1 <= 0; t1++)
  for (t2 = 0; -511 + t2 <= 0; t2++) {
    A2[4096*t1+8*t2] = 0;
    for (t3 = 0; -511 + t3 <= 0; t3++)
      A2[4096*t1+8*t2] = A2[4096*t1+8*t2] ⊙ (A1[4096*t1+8*t3] ⊙ A3[8*t2+4096*t3]);
  }

```

Fig. 5. Matrix multiply after scalar to array conversion.

Applying this algorithm on our example results in the code shown Figure 5. Notice that no scalar reference remains: the refactored code causes less dependencies and is easier to handle for source-to-source parallelizers.

Notice that none of the presented transformations should be normally required if source-to-source parallelizers would be more robust. Note also that our techniques cannot be used to remove all scalar references, due to overly complex data flows. However we observed that our approach is usually sufficient to remove most of the inessential scalar references.

Once these code transformations have been performed, the resulting C code is simply sent to the back-end parallelizer which parallelizes it.

3.3. Reverting the Outlining

The output of the back-end parallelizer is a parallel C code where every operation is hidden in $\odot$. Before compiling the code back to a x86-64 binary, we revert this outlining to restore the correct semantics in the parallel code.

One could simply consider replacing each C instruction in the parallelized loop nests by the corresponding assembly code extracted from the binary program. This would be wrong since the induction variable resolution leads to replace some registers with equivalent affine functions. Thus, the dependence analysis is performed on the code without those register references, and we cannot guarantee a correct semantics if we directly re-inject them. Instead, the registers are represented as equivalent thread-local C variables, and their definitions are evaluated in pure C. The actual operations on these registers are expressed as inlined assembly code, except for SIMD instructions, which are re-injected using SIMD compiler intrinsics. For example, if `rax` has been determined as being equal to $1024 + 8 \times i$, then the instruction `mov rbx, rax`, is replaced by the following code:

```

long int rbx, rax = 1024 + 8 * i;
__asm__ ("mov %0, %1", rbx, rax);

```

The other register SSA versions are also represented as thread-local C variables. If a register version is alive at loop entry, the hardware register is used to initialize the corresponding variable. Otherwise, we try to rebuild the value of this register version as an affine combination of the alive register versions and memory elements.

An equivalent problem also occurs for registers updated during the loop nest execution and used after it has completed. As the code has been transformed, it is generally impossible to ensure that the variable representing such register version contains the correct value at loop exit. The main exceptions are the register versions handled through scalar evolution. In those cases, the correct value of the register at loop exit can be easily computed using common polyhedral tools such as PIP [Feautrier 1988], when the loop nest is affine.

In we are not able to transmit the value of a register version to or from its equivalent C variable, our system cancels the parallelization of the concerned loop nest.

We show in Figure 6 the matrix multiply code after transformation, parallelization and lowering. One can see that the initialization of the result matrix has been split out

```

#pragma omp parallel for private(t1,t2,t3,t4)
for (t1=0; t1<=15; t1++)
  for (t2=0; t2<=15; t2++)
    for (t3=32*t1; t3<=32*t1+31; t3++)
      for (t4=32*t2; t4<=32*t2+31; t4++) {
        *((double*)(10494144+4096*t3+8*t4))=0;
      }
#pragma omp parallel for private(t1,t2,t3,t4,t5,t6,xmm0v1,xmm0v2)
for (t1=0; t1<=15; t1++)
  for (t2=0; t2<=15; t2++)
    for (t3=0; t3<=15; t3++)
      for (t4=32*t1; t4<=32*t1+31; t4++)
        for (t5=32*t2; t5<=32*t2+31; t5++)
          for (t6=32*t3; t6<=32*t3+31; t6++){
            xmm0v1 = _mm_load_sd((double *) (6299808 + 4096*t4 + 8*t6));
            __m128d tmp1 = _mm_load_sd((double*) (8396960+8*t5+4096*t6));
            xmm0v2 = _mm_mul_sd(xmm0v1, tmp1);
            __m128d tmp2 = _mm_load_sd((double*) (10494144+4096*t4+8*t5));
            tmp2 = _mm_add_sd(tmp2, xmm0v2);
            _mm_store_sd((double *) (10494144+4096*t4+8*t5), tmp2);
          }

```

Fig. 6. Matrix multiply after transformation by PLUTO and semantics restoration.

of the main loop nest by PLUTO. Some C variables, representing hardware registers SSA versions, are used in addition to some SIMD intrinsic functions. The parallelization is achieved through simple OpenMP [openmp 2008] directives. No register variable needs to be initialized here and none is alive after the loop, thus no initialization or termination code appears.

3.4. Runtime Component

The transformed loop nest source codes are individually stored in separate functions. Those functions are compiled in a dynamic library using a standard C compiler such as GCC. A runtime component, automatically generated, is in charge of rerouting the execution flow towards the transformed loop nests. One could easily avoid using this runtime component, replacing it with a static binary rewriter which could inject the parallelized loop nests directly in the original binary. As the overhead of our runtime component is low (mainly two system calls per loop nest execution), there is no best solution here.

At startup, just before calling the application entry point, the runtime component inserts a breakpoint at the beginning of every loop nest which has been parallelized. Then, the runtime component launches the application and stalls. When a breakpoint is reached, the runtime component stops the program execution. The hardware register values in the application execution context are made available to the transformed code, in order to initialize the C variables representing the registers. Then, the runtime component reroutes the program towards the corresponding parallel loop nest function. When the parallel loop nest execution has completed, the runtime system is awoken, and redirects the execution to the end of the loop nest in the original program.

4. EVALUATION

We have evaluated our implementation on the PolyBench suite of programs [polybench 2010]. This suite is made of kernels commonly used in scientific and multimedia codes. The goal of this section is to provide answers to three basic questions one may want to ask about static parallelization of binary code.

Table I.

Program	Clang				GCC				ICC			
	-O2		-O3		-O2		-O3		-O2		-O3	
2mm	6/8	75%	6/8	75%	6/8	75%	6/8	75%	18/21	86%	10/23	43%
3mm	9/11	82%	9/11	82%	9/11	82%	9/11	82%	26/29	90%	15/33	45%
atax	6/7	86%	6/7	86%	5/7	71%	5/7	71%	5/7	71%	5/7	71%
bicg	5/6	83%	5/6	83%	5/6	83%	5/6	83%	4/6	67%	4/6	67%
correlation	10/13	77%	10/13	77%	8/13	62%	8/14	57%	21/27	78%	19/25	76%
covariance	11/13	85%	11/13	85%	11/13	85%	11/16	69%	16/22	73%	16/23	70%
doitgen	10/13	77%	10/13	77%	10/13	77%	10/13	77%	9/14	64%	9/15	60%
gemm	9/11	82%	9/11	82%	9/11	82%	9/11	82%	11/14	79%	4/12	33%
gemver	9/10	90%	9/10	90%	9/10	90%	9/10	90%	15/17	88%	11/13	85%
gramschmidt	8/10	80%	8/10	80%	7/10	70%	7/10	70%	8/11	73%	8/11	73%
jacobi-2d	7/9	78%	7/9	78%	7/9	78%	7/9	78%	5/10	50%	5/10	50%
lu	6/8	75%	6/8	75%	4/7	57%	4/7	57%	9/14	64%	9/14	64%
	81%		81%		76%		74%		74%		61%	

- How many loops is the decompilation phase able to extract? Does the use of binary code entail any loss in coverage? What is the effect of compiler optimizations on coverage?
- Does the use of binary code entail any loss in performance compared to the use of source code? Does this depend on the power of the parallelization back-end?
- How does performance speedup obtained by automatic binary code parallelization compare to that of hand-made, directive-based parallelization? To other published systems?

The next three sections focus on each of these questions in turn, using an Intel Xeon W3520 processor with four cores and two threads per core running Linux as the testing platform.

The reader should be warned of an easily missed characteristic of the PolyBench programs: they include a main *kernel*, along with an *initialization* loop nest which performs a significant amount of work and is trivially parallelizable. Even though the intent is clearly that only the kernel part be used for benchmarking, some researchers have included the initialization loop(s) in evaluation runs. In the following experiments, we mention explicitly which part of the programs we use.

4.1. Loop Coverage

To evaluate the quality of the decompilation, we have used six different combinations of compiler and optimization level. The compilers involved are Clang 2.8 from the LLVM suite, GCC 4.5, and the Intel ICC compiler version 12.1, each at -O2 and -O3 optimization levels. The resulting executable programs were submitted to the static binary code decompilation described in Section 2. In this subsection, the whole program has been analyzed.

In all cases, all loops have been correctly *located* in the binary code. However, not all loops are *usable*. We define the coverage of our system as the number of loops that can be safely transmitted to the parallelization back-end. We have chosen to *not* make use of “background knowledge” about library function calls e.g., a single call to `sqrt` in the main loop of the *correlation* program makes the outer loop unparallelizable. This has to be contrasted with source-to-source tools that require explicit pragmas (e.g., PLUTO), where the programmer is supposed to leave only “harmless” calls, which are then ignored.

Table I shows counts and percentages for every combination of compiler and optimization level. Each column shows the ratio between the number of loops that can

safely be transmitted to the parallelization back-end and the total number of loops located in the binary code, the corresponding percentage, with an average at the bottom of the column.

There are two main causes for abandoning a loop. The first is the presence of function calls: all benchmarks include a final loop printing the computed data structure, usually a two or three-dimensional matrix. This accounts for the vast majority of loops ignored in Clang and GCC binaries (the remaining failures are caused by the difficulty of tracking register values across calls without making simplifying assumptions).

The second major source of unparallelizable loops is the use of sophisticated program transformations. As appears from the numbers in Table I, ICC makes heavy use of loop transformations, at both -O2 and -O3 levels. In the worst case (2mm at -O3), less than half of the loops remain usable. This is due to loop tiling, which ICC seems to automatically apply. The problem here is that tiling produces non-strictly linear address expressions, e.g., using `max` to compute bounds on “side-tiles”. Even though we could detect such cases, we think a more general loop restructuring phase is needed to handle highly optimized codes. We have left this for future research.

4.2. Binary-to-Binary vs. Source-to-Source

The goal of our second set of experiments is to evaluate whether parallelization applied to binary code were fundamentally inferior to automatic parallelization of the source code. Since the core of our binary parallelizer uses a source-to-source back-end parallelizer (acting on a skeleton program extracted from the binary), we compare the automatic parallelization of (1) the source code and (2) the skeleton code extracted from the binary. The experiments in this case is applied to the kernel part only.

We have selected two source-to-source parallelizers, CETUS 1.3 and PLUTO, to act as back-end components. Each of them has been used to (1) generate parallel versions of the source code and (2) generate parallel version of the loops extracted from the binary code generated with `gcc -O2`. In both cases, the final executable is generated with `gcc -O3`.

The resulting speedups are shown on Figure 7, with the base execution time being the sequential execution of the program. The goal here is not to compare both parallelizers, since they use significantly different strategies, but rather compare the speedups obtained on source code and on binary code.

The behavior of both back-end parallelizers is clearly different. In the case of PLUTO, performance varies slightly between both versions in most of the cases. However, these variations can hide large differences in the nature of transformations applied. For instance, binary-to-binary slightly outperforms source-to-source for `correlation`, even though fewer loops have been parallelized in the binary case (because of the presence of a call to `sqrt`). Conversely, the presence of a call in `gramschmidt` completely annihilates performance in the binary case, whereas on source code PLUTO simply assumes that the call may not interfere with the rest of the loop nest. The only case where the difference is significant is `gemver`, where failure on scalar expansion has obviously prevented PLUTO from finding a transformation that it found in the source code.

The case of CETUS is also interesting, in that it doesn't find anything profitable to do on source programs. This is due to the timing functions present in the benchmark programs which make the alias analysis of CETUS 1.3 fail (the CETUS developers have been informed and will probably correct the weakness in the next version of the software). In some cases this has been bypassed when using binary code directly, because the analyzable loop nests are extracted from the rest of the program by our system (in 2mm, 3mm, doitgen, and gemm). In other cases, the loop nests were simply too complex and required parallelizing transformations, which were out of reach of CETUS in its current state. Nevertheless, this experiment is not designed to serve as

Fig. 7. Parallelization back-ends applied to (1) source code, and (2) binary code (kernel only).

a comparison between PLUTO and CETUS, only to evaluate their respective role as back-end compilers.

4.3. Binary-to-binary vs. Hand-Parallelization

The goal of our last experiment is to compare our solution first to a skilled programmer using OpenMP directives, and second to the best results achieved by a published automatic binary parallelizer (as far as we know). The four contenders are:

- an unnamed programmer using OpenMP, placing `parallel` for directives on the outermost parallel loop of each loop nest (without transformation);
- our binary parallelization system with PLUTO (i.e., performing code transformations);
- our binary parallelization system with CETUS 1.3 as a back-end (no transformation);
- the results published by Kotha and colleagues [Kotha et al. 2010] (no transformation).

Numbers in the last category have been taken directly from the paper Kotha et al. [2010, Section VIII, Table II] when available, using experiments on an architecture similar to the one we have used. No attempt has been made to reproduce their system (see their Acknowledgments section). The various speedups are shown on Figure 8.

An important aspect of this experiment is that parallelization is applied to the whole program and not only on the kernel loop nests. For all experiments, all loops in sight have been parallelized when possible, including initialization loops (see the discussion of the PolyBench programs at the beginning of this section). The reader may wish to compare the strong impact of including initialization by comparing speedups shown on Figure 7 with those of Figure 8, e.g., on *atax* or *bicg*. We would like to state here that we do not consider whole-program benchmarking to be a meaningful way to evaluate parallelization systems on the PolyBench suite; we show these results for the sake of comparison with other systems.

Fig. 8. Comparing simple, hand-made OpenMP parallelization with static automatic binary parallelizers (initialization+kernel).

Among the four parallelizers used, only the PLUTO back-end is able to apply code transformations, giving it a significant advantage on some programs. We have kept the polyhedral back-end in this set of results to illustrate the modularity of our system, where a single run-time switch can provide a large gain.

These results can be roughly divided into four categories:

- (1) 2mm, 3mm, and gemm, where all non-transforming systems obtain similar results, and polyhedral transformations lead to a spectacular gain; atax could be included in this category, except that polyhedral techniques cannot compensate for the fact that the only parallel loop is buried inside a non-parallel outer loop;
- (2) covariance and correlation, where polyhedral techniques also dominate clearly, although at a smaller scale: The locality optimization allowed by the polyhedral model has a strong impact, compensating in both cases for the fact that some loops were not parallelizable (due to function calls);
- (3) gemver, gramschmidt, and lu, where the presence of function calls and the complex expansion and privatization requirements make most automatic system fail (the system described by Kotha et al. [2010] seems to perform very well on gemver, but results on the last two programs have not been published);
- (4) big, doitgen, and jacobi, where our automatic system is clearly suboptimal compared to manual, OpenMP parallelization (for reasons similar to the previous category), and where both of these are far below the system by Kotha and colleagues. We have to admit that we have no explanation for this last fact: despite our best efforts (including hand tweaking), we have not been able to even approach the results published in Kotha et al. [2010], especially on jacobi, a stencil-like kernel.

```

for (t1 = 0; t1 <= rsi; t1++)
  for (t2 = 0; t2 <= rdi; t2++) {
    M[rax + 8*t1] = ...
    ...
    M[rbx + 8*t1 + 8*t2] = ...
  }

```

Fig. 9. Sample code with parametric loop bounds and access functions.

Overall, our conclusion on this diverse set of experiments is first that our automatic binary parallelization competes with equivalent systems in most cases, and second that the ability to use polyhedral techniques can make a significant difference in some cases.

5. DISCUSSION ON CODE COMPLEXITY

Our system performs well on the kernels of the PolyBench benchmark suite. Those kernels are however not representative of all existing programs. We suggest in this section some directions that may enable to handle a wider range of programs in the future. The presented measurements have been made in the same environment as in the preceding section.

5.1. Extension to Parametric Codes

The programs tested in Section 4 all contain non-parametric loop bounds and access functions. However, a developer can choose to use parametric loop bounds or dynamic arrays, leading to registers (parameters) appearing in the code extracted from the binary code. There is no strong restriction preventing our system to handle these cases, because the polyhedral model and its associated tools are able to solve parametrized problems. Dependence analysis is however strongly conservative when parameters appear in access functions, and nearly always assumes that two such memory references can access the same data.

To unleash some parallelism, our system makes some assumptions enabling it to split the accessed memory into distinct arrays, which are assumed to not intersect by the dependence analyzers.

In the case of parametric loop bounds but non-parametric access functions, Kotha et al. [2010] have suggested to split memory accesses by assuming that the constant value in access functions are the array base addresses. Then runtime checks ensure the validity of the memory partitioning. When access functions contain parameters (registers), a similar technique can be applied.

For example, one can see in Figure 9 a simple loop nest with two memory accesses. As both memory accesses use different registers as base value (rax and rbx), they are assumed to never intersect. The code is simplified accordingly: our system removes the base registers after renaming the accessed arrays. Using existing polyhedral tools, one can easily compute minimum and maximum values of a linear expression, even in presence of loop indices. This allows our system to automatically generate the tests devoted to check at runtime if memory references assumed to point to different arrays are actually independent. One can see in Figure 10 the resulting code: The memory has been split into distinct arrays, and the registers in the access functions have been removed. The tests to ensure that both memory accesses do not intersect are generated before the loop nest. At runtime, if the test fails, the runtime component executes the original sequential loop nest and does not consider this optimized version anymore.


```

if (rax + 8*rsi >= rbx && rax <= rbx) return FAIL;
if (rbx + 8*rsi + 8*rdi >= rax && rbx <= rax) return FAIL;
for (t1 = 0; t1 <= rsi; t1++)
  for (t2 = 0; t2 <= rdi; t2++) {
    A0[8*t1] = ...
    ...
    A1[8*t1 + 8*t2] = ...
  }

```

Fig. 10. Sample parametric code with simplified memory references.

Sequential	Source parallelization	Binary parallelization
47.06 s	30.76 s (1.5)	40.79 s (1.2)

Fig. 11. Execution times (and speedups) for swim on the train dataset.

A set of tests is generated for each pair of memory accesses where at least one access is a write. This could lead to generate many tests, but in practice the tests, placed outside of any loop, do not cause any significant overhead.

This technique allows us to handle codes like the *swim* code from the SPEC OMP-2001 benchmark suite [SpecOMP 2001]. As our scalar removal process cannot efficiently handle this program, polyhedral transformation of this code is not yet possible. However our system is still able to parallelize it using non-transforming parallelization techniques. One can see on Figure 11 that our system is able to perform a reasonable speedup compared to the speedup obtained from the reference source code parallelization. It is important to remember here that our system is automatic and that it does not need the source code, whereas the reference speedup is reached after human intervention to provide hints to the compiler.

5.2. Extension to Polynomial Codes

Another common issue in binary codes is array linearization. The compiling process that transforms multi-dimensional array accesses into flat memory access functions often yields non-linear expressions. Those expressions cannot be handled by the polyhedral model. Unfortunately they are frequent in common binary codes, especially in Fortran programs where dynamic arrays are commonly used.

In order to parallelize such programs, we propose to use an approximate and conservative dependence analysis method handling parametrized polynomial memory references. We denote by $r_k(I_k, P_k)$ a memory reference where I_k is the list of variables, and P_k the list of parameters used in the reference. Variables I_k are the indices of the loops enclosing the memory reference. Those loop indices are constrained by the loop bounds: their possible values define a convex polyhedron $\mathcal{D}_k$, the iteration domain of the memory reference r_k . We note $I_k[d]$ the index of the loop at depth d in the hierarchy of the loops enclosing the memory reference. Parameters P_k are usually array-size parameters whose values are stored in processor registers.

Consider two memory references $r_1(I_1, P_1)$ and $r_2(I_2, P_2)$ inside a loop nest. At least one of those accesses is a write, and both $r_1(I_1, P_1)$ and $r_2(I_2, P_2)$ are non-linear parametrized expressions, more precisely multivariate parametrized polynomials. If we are able to ensure that those memory accesses do not provoke any dependency between iterations of the outermost loop, then this loop can be parallelized. A sufficient condition is the statement:

$$\begin{aligned} &\text{for any } I_1 \in \mathcal{D}_1 \text{ and } I_2 \in \mathcal{D}_2 \text{ such that } I_1[0] \neq I_2[0] \\ &r_1(I_1, P_1) < r_2(I_2, P_2) \text{ or } r_1(I_1, P_1) > r_2(I_2, P_2) \end{aligned} \quad (1)$$

Sequential	Source parallelization	Binary parallelization
89.29 s	28.98 s (3.1)	48.46 s (1.8)

Fig. 12. Execution times (and speedups) for `mgrid` on the train dataset.

This condition ensures that there is no intersection between the values reached through $r_1(I_1, P_1)$ and $r_2(I_2, P_2)$ across distinct iterations of the outermost loop, and thus, that those memory accesses do not induce any dependence carried by the outermost loop. Ensuring (1) for every couple of memory accesses where at least one is a write allows us to parallelize the outermost loop. Notice that r_1 and r_2 can be a single memory write to handle self-dependences. Very little information or even nothing is known about the parameters values. Hence (1) can only be satisfied subject to some conditions on the values of the parameters.

To find some sufficient conditions on the parameter values ensuring that (1) is satisfied, we use a method based on symbolic Bernstein expansion of polynomials defined over parametrized convex polytopes, described in Clauss and Tchoupaeva [2004] and Clauss et al. [2009] and implemented in the ISL library [Verdoolaege 2010]. This method allows us to compute the maximum value that can be reached by a polynomial. Hence when the maximum value of $r_1(I_1, P_1) - r_2(I_2, P_2)$ (respectively $r_2(I_2, P_2) - r_1(I_1, P_1)$) is strictly negative, we obviously prove that $r_1(I_1, P_1) < r_2(I_2, P_2)$ (respectively $r_1(I_1, P_1) > r_2(I_2, P_2)$).

Consider the following example built from one of the handled binary codes:

$$\begin{cases} r_1(i_1, j_1, k_1, M, N, P) = 8 + P + 8M + 8N + 8i_1N + 8j_1M + 8k_1 \\ r_2(i_2, j_2, k_2, M, N, Q) = 8 + Q + 8M + 8N + 8i_2N + 8j_2M + 8k_2 \\ 0 \leq i_1, i_2 \leq L - 1, 0 \leq j_1, j_2 \leq L - 1, 0 \leq k_1, k_2 \leq L - 1 \\ i_1 \neq i_2, L > 2, M > 0, N > 0, P > 0, Q > 0 \end{cases}$$

Using the ISL library, we automatically compute the sufficient condition statements:

if $((-8 + 8L)M + (-8 + 8L)N + 8L + P - Q - 8 < 0$ **or** $(-8 + 8L)M - 8N + 8L - P + Q - 8 < 0)$
and $((-8 + 8L)M - 8N + 8L + P - Q - 8 < 0$ **or** $(-8 + 8L)M + (-8 + 8L)N + 8L - P + Q - 8 < 0)$
then there is no dependency between r_1 and r_2 carried at level 0

Similar tests are generated for every couple of memory accesses where at least one is a write. Those tests are evaluated at runtime and, if one fails, the parallelization of the outermost loop cannot be proven correct and the sequential version of the loop nest is run.

We have implemented this strategy: it enables our system to parallelize the `mgrid` code from the SPEC OMP benchmark suite. Figure 12 shows the execution times and speedups reached by our system compared to the reference parallelization. Once again, we consider this as a good result, considering that our automatic system does not need the source code and that few common parallelization techniques can be applied in this case, since the access functions are not linear.

6. RELATED WORK

6.1. The Polytope Model

The polytope model is a well-established theory [Feautrier 1992; Clauss and Loechner 1998; Bastoul 2004] and many tools are available to perform different computations in this framework. Some automatic parallelizing compilers, taking advantage of this model, are also being developed such as PLUTO [Bondhugula et al. 2008] or PoCC/LetSee [Pouchet et al. 2010]. These compilers take source code as input, where

the interesting loop nests have been hand-marked with compiler directives. They are able to automatically analyze these loop nests, perform an optimizing transformation, and generate parallel loop nests containing standard parallel control structures (e.g., OpenMP directives). Our system can be seen as an interface between raw binary code and these compilers: it is able to rebuild well-formed C code allowing these tools to handle it, and then to re-inject raw instructions in order to recompile the transformed nests.

6.2. Parsing Binary Code

Any system working on binary code needs to somehow parse this code and extract usable structures. There is however a large spectrum between the static analysis of binary code (as illustrated in this paper) and the dynamic optimization of a running program (a more common situation). Thread level speculation (TLS) is currently the major category of systems that have to deal with low-level, executable code, and it is interesting to look at how they deal with it. A typical epitome of TLS is the POSH system [Liu et al. 2006], where code is statically analyzed to extract tasks and where the runtime environment is responsible for verifying the absence of conflicts. In the words of the authors [Liu et al. 2006, Introduction]: “[TLS] compilers are unique in that they do not need to fully prove the absence of dependences across concurrent tasks—the hardware will ultimately guarantee relevant to a comparison to our work, DeVuyst et al. [2011, Section 3] write: “Without the high-level code, we cannot guarantee that parallel iterations of [a] loop will not attempt to modify the same data in memory.” These two short quotations are sufficient to highlight what makes our work original regarding binary code parsing.

Where TLS systems focus on run-time checking for dependencies, relying on some sort of transactional memory to backtrack in case of a conflict [Hertzberg and Olukotun 2011], binary parallelization systems simply produce a different program off-line, which means that they need some precise (and definitive) characterization of dependences right out the binary code. Therefore, most TLS systems perform control-flow analysis, extracting routines and loops [Liu et al. 2006; DeVuyst et al. 2011]. Some go as far as putting the program into SSA form to perform a simple, local form of data-flow analysis [Yang et al.; Ootsu et al. 2002]. However, we know of no system that builds symbolic expressions for memory accesses as we do. Static binary parallelization needs this description to perform static dependence analysis, whereas TLS systems leave that part to the run-time environment and/or hardware. On the other hand, binary parallelization systems restrict themselves to loop nests, whereas TLS systems target a much wider range of program structures, including, unfortunately, loop nests that could be handled statically.

6.3. Binary Rewriters

Many binary rewriters have been developed, e.g., PLTO [Schwarz et al. 2001], DIABLO [Van Put et al. 2005], or PEBIL [Laurenzano et al. 2010]. These tools are able to parse binary programs, and to perform some optimizations directly on binary files. However, as far as we know, these tools have specific application domains, and none of them deals with parallelization of high-level structures like loop nests. Conversely, each of them could potentially be used as a binary code manipulation API in our system.

6.4. Parallelization of Binary Code

To our knowledge, only two binary parallelizers not performing address speculation have been developed recently. First, Yardımcı and Franz [2006] propose the first attempt at binary parallelization. Their system identifies control flow that behaves like loops at runtime, including recursive function calls. It is then able to parallelize or

vectorize these code slices. As the system is highly dynamic, it can handle more complex loop structures, however they cannot rely on complex decision algorithms. Therefore, it is restricted to parallelize loops where no data dependence occurs, strongly limiting its scope.

Recently, Kotha and colleagues have proposed a static binary parallelizer [Kotha et al. 2010]. We have already mentioned in Section 4 how the evaluation methodology used in that paper is strongly biased by whole-program speedup measures. However, this work is highly similar to ours, and the rest of this section compares their approach to ours.

The first major difference is how both systems extract address expressions. Whereas their system uses simple pattern matching to recognize counter initialization, test, and increment for each loop, our system captures the data-flow of address computations, and uses symbolic analysis to reconstruct address expressions built around normalized loop counters. Although the naive approach appears sufficient on PolyBench programs, we have found it inefficient on more complex loops, like the ones our decompilation pass extracts from SPEC benchmarks (see Section 5). There are two major reasons why a more subtle analysis of binary code is desirable. The first is that most programs put enough pressure on the register allocator to force spilling on registers containing loop counters, if any (this is even true on PolyBench programs compiled on a 32-bit architecture). The second is that dependence analysis needs loop trip-counts, which are seldom expressed as single compare instructions. The approach we have developed in Section 2.4 is, in our experience, absolutely crucial to obtain reasonable constraints on blocks and avoid an overly conservative dependence analysis.

The second major difference is the parallelization strategy. Kotha et al. [2010] use several simple dependence tests to decide whether a given loop is parallel or not. The dependence vectors are extracted with ad-hoc pattern-matching techniques similar to the ones used in extracting addresses. A trivial analysis of the dependence vector components selects parallel loops. This strategy is known to be fundamentally inferior to polyhedral techniques, where each dependence is represented as a polyhedron, an abstraction strictly more expressive than dependence vectors. This lets our system apply sophisticated transformations to produce parallel versions of the code, instead of simply flagging parallel loops.

7. CONCLUSION AND FUTURE WORK

In this paper, we present an automatic system, able to perform advanced parallelizing transformations on binary codes. We show on a set of kernels that the generated codes have similar performance as those parallelized from the source.

The described decompilation phase is more specifically targeting polyhedral compilers. It allows our system to perform advanced code transformations but it also restricts its scope, since a very precise description of the binary programs is required.

Those two aspects are competing and should be handled differently to further extend our work. We plan to perform only partial raising of some loop nests. For some complex codes, raising only memory accesses and loop bounds while ignoring the control flow inside the outermost loop would allow us to parallelize them. This partial raising would however restrict the set of parallelizing transformations that could be applied on the code.

On the opposite, some codes are totally analyzable and we plan to extend our work by applying even more aggressive optimizations on them. For example, performing a full decompilation would allow us to recompile the loop nests for different architectures, resulting in a binary translation system able to generate optimized parallel code for GPUs or FPGAs from sequential binary x86 programs.

REFERENCES

- APPEL, A. AND GINSBURG, M. 2004. *Modern Compiler Implementation in C*. Cambridge University Press.
- BAE, H., BACHEGA, L., DAVE, C., LEE, S.-I., LEE, S., MIN, S.-J., EIGENMANN, R., AND MIDKIFF, S. 2009. Cetus: A source-to-source compiler infrastructure for multicores. In *Proceedings of CPC '09*.
- BASTOUL, C. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of PACT'04*. IEEE Computer Society, Los Alamitos, CA.
- BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of PLDI '08*. ACM, New York, USA.
- CLAUSS, P., FERNÁNDEZ, F. J., GARBERVETSKY, D., AND VERDOOLAEGE, S. 2009. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Trans. VLSI Syst.* 17, 983–996.
- CLAUSS, P. AND LOECHNER, V. 1998. Parametric analysis of polyhedral iteration spaces. *J. VLSI Signal Process. Syst.* 19, 179–194.
- CLAUSS, P. AND TCHOUPAEVA, I. 2004. A symbolic approach to Bernstein expansion for program analysis and optimization. In *Proceedings of CC'04*. Springer.
- CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Long. Syst.* 13, 4, 451–490.
- DEVUYST, M., TULLSEN, D. M., AND KIM, S. W. 2011. Runtime parallelization of legacy code on a transactional memory system. In *Proceedings of HiPEAC '11*. ACM, New York, NY.
- FEAUTRIER, P. 1988. Parametric integer programming. *RAIRO Recherche Opérationnelle* 22, 3, 243–268.
- FEAUTRIER, P. 1992. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *Int. J. Parall. Program.* 21, 313–347.
- HERTZBERG, B. AND OLUKOTUN, K. 2011. Runtime automatic speculative parallelization. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*. 64–73.
- KOTHA, A., ANAND, K., SMITHSON, M., YELLAREDDY, G., AND BARUA, R. 2010. Automatic parallelization in a binary rewriter. In *Proceedings of MICRO '43*. IEEE Computer Society, Los Alamitos, CA.
- LAURENZANO, M. A., TIKIR, M. M., CARRINGTON, L., AND SNAVELY, A. 2010. PEBIL: Efficient static binary instrumentation for linux. In *Proceedings of ISPASS '10*.
- LIU, W., TUCK, J., CEZE, L., AHN, W., STRAUSS, K., RENAU, J., AND TORRELLAS, J. 2006. Posh: a tls compiler that exploits program structure. In *Proceedings of PPOPP '06*. ACM, New York, NY, 158–167.
- MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- OOTSU, K., YOKOTA, T., ONO, T., AND BABA, T. 2002. Preliminary evaluation of a binary translation system for multithreaded processors. In *Proceedings of the Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*. 77–84.
- openmp 2008. The OpenMP API specification. <http://www.openmp.org>.
- polybench 2010. Polybenchs 1.0. <http://www-rocq.inria.fr/pouchet/software/polybenchs>.
- POUCHET, L.-N., BONDHUGULA, U., BASTOUL, C., COHEN, A., RAMANUJAM, J., AND SADAYAPPAN, P. 2010. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the Conference on Supercomputing (SC '10)*. IEEE Computer Society Press, Los Alamitos, CA.
- SCHWARZ, B., DEBRAY, S., ANDREWS, G., AND LEGENDRE, M. 2001. Plto: A link-time optimizer for the intel ia-32 architecture. In *Proceedings of the Workshop on Binary Translation*.
- SpecOMP 2001. SPEC OMP benchmark suite. <http://www.spec.org/omp/>.
- VAN PUT, L., CHANET, D., DE BUS, B., DE SUTLER, B., AND DE BOSSCHERE, K. 2005. Diablo: A reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the IEEE Symposium on Signal Processing and Information Technology*.
- VERDOOLAEGE, S. 2010. ISL: An integer set library for the polyhedral model. In *Proceedings of ICMS*.
- VERDOOLAEGE, S., SEGHIR, R., BEYLS, K., LOECHNER, V., AND BRUYNNOOGHE, M. 2007. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica* 48, 1, 37–66.
- YANG, J., SKADRON, K., SOFFA, M.-L., AND WHITEHOUSE, K. Feasibility of dynamic binary parallelization. http://www.usenix.org/event/hotpar11/tech/final_files/Yang.pdf.
- YARDIMCI, E. AND FRANZ, M. 2006. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proceedings of CF '06*. ACM, New York, NY, 127–138.

Received July 2011; revised October 2011, November 2011; accepted December 2011