Transparent Parallelization of Binary Code

Benoît Pradelle Alain Ketterlin Philippe Clauss

CAMUS group, INRIA Nancy Grand Est and LSIIT, Université de Strasbourg, France.
{pradelle, ketterlin, clauss}@icps.u-strasbg.fr

ABSTRACT

This paper describes a system that applies automatic parallelization techniques to binary code. The system works by raising x86-64 raw executable code to an intermediate representation that exhibits all memory accesses and relevant register definitions, but outlines detailed computations that are not relevant for parallelization. It then uses an off-the-shelf polyhedral parallelizer, first applying appropriate enabling transformations if necessary. The last phase lowers the internal representation into a new executable fragment, re-injecting low-level instructions into the transformed code. The system is shown to leverage the power of polyhedral parallelization techniques in the absence of source code, with performance approaching those of source-to-source tools.

Keywords

Static parallelization, Binary code, Polytope model

1. INTRODUCTION

Due to the physical limits recently reached, improvements in processors' clock speed has been stopped. Processors manufacturers are now increasing the number of cores on processors, putting an important pressure on parallelism extraction. Many automatic parallelization techniques have been proposed in the past few decades. A vast majority of those concentrate on parallelism extraction from source code; however in many cases it is interesting to consider parallelizing applications after their compilation, at the binary level. This approach allows one to handle any application independently from its source language, even if the source code has been lost (legacy code) or if the source code is not distributed (closed source software). Moreover with this approach, the whole application can be parallelized, including invoked libraries. Thus, the presented solution can be implemented as an OS service able to automatically parallelize any binary application, typically during its installation.

Analyzing a binary application can be a complex task: it might be difficult to extract even a simple control flow in some cases. One could propose to wait for the runtime to benefit from a totally determined execution context and perform an exact analysis. However we observed that static analysis is enough in many cases and does not suffer from overheads induced by dynamic techniques.

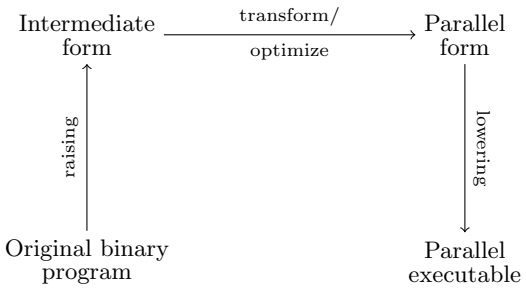

Figure 1: The system's architecture

To perform an efficient parallelization, binary code needs to be raised to some more suitable representation. Typically, one could think of some classical compiler intermediate representations such as the ones used by LLVM [1] or GCC. However those representations are usually unsuited to describe assembly code as is (with non-typed data, one unique array to represent the memory, actual register allocation, ...). Moreover using an internal representation imposes to implement the needed parallelization strategies for this specific representation. We have chosen to use C code as intermediate representation and only raise a small subset of the code semantic. Source to source parallelizers working with C code can then be used to obtain a parallel intermediate form, into which the detailed semantic is restored and then compiled to obtain a parallel executable.

We concentrate on extracting loop nests fitting the polytope model in order to benefit from the powerful transformations performed in this model.

The basic architecture of our system is shown on Figure 1. First, the x86-64 binary code of the program is parsed and transformed into an intermediate representation whose concrete syntax is C. Second, the resulting C code is submitted to a parallelizing compiler, whose role is to detect parallel loops and which is free to apply any program transformation. Third, the resulting parallel code is transformed back into executable code, and embedded in a lightweight run-time component that is

responsible for replacing original regions of code by new, optimized ones.

2. EXTRACTING PARALLELIZABLE REGIONS FROM BINARY CODE

The first phase consists in parsing the binary code of the program, extracting one routine at a time, and building a control-flow graph (CFG) for the routine whenever possible. Indirect branches are tentatively solved by scanning jump tables; no optimization will ever be attempted on routines where the CFG is not fully recovered. After the dominator tree is computed, a loop hierarchy is reconstructed using DJ-graph [10] based techniques (irreducible loops are "solved" by replicating parts of their bodies). At the end of this phase, every loop has a unique entry block, a set of local basic blocks, and a set of sub-loops. Note that loops are reconstructed as they appear in the binary, and no refactoring takes place. If the compiler has unrolled and/or peeled some iterations, the original iteration domain of the loop may not be fully recovered. We have left such refactoring for future research.

The next step is to perform a data-analysis of the program, by putting it into static single assignment (SSA) form. This gives a unique version number for each register definition, and provides a direct use-def link for each register usage. All registers are tracked, and memory is handled as a weakly updated variable (where each memory write explicitly refers to the currently visible version). Starting from every memory access, and "slicing back" through use-def links, it is possible to reconstruct the computation of the memory access' address. The slicing proceeds while the reconstructed function is linear, and until meeting either a memory access that cannot be resolved, or an input register (*i.e.*, a routine parameter), or a ϕ-function. The result is a linear combination of registers. Whenever a ϕ-function appears in a memory access function, an induction-variable resolution procedure tries to express it as a linear function of a newly introduced normalized loop counter, and in cases where this succeeds, the register is replaced by the resulting expression. In many cases, this sequence of steps is enough to express all memory accesses inside one or more loops as linear expressions involving loop counters and loop-invariant registers, all with integer coefficients.

Similarly, branch conditions are tentatively expressed as linear expressions compared to zero with $<$, $\leq$ or $=$ (and their negations). Combined with control dependence (which can be computed from the CFG) on blocks that may exit the various loops involved, one can compute loop "trip-counts" or, more precisely, symbolic constraints on blocks inside loops. Here again, these constraints involve linear combinations of loop counters and loop-invariant registers: they are used later as guards when generating C code from the binary, and also by the parallelizing compiler to decide which polyhedral transformation to apply. Before that, a final slicing step starts from all linear expressions involved (in memory accesses and branch conditions) and selects which

```
for (t1 = 0; -1023 + t1 <= 0; t1++)
   for (t2 = 0; -1023 + t2 <= 0; t2++) {
      M[23371872+8536*t1+8*t2] = 0;
      xmm1 = 0;
      for (t3 = 0; -1023 + t3 <= 0; t3++) {
         xmm0 = M[6299744+8296*t1+8*t3];
         xmm0 = xmm0 ⊙ M[14794848+8*t2+8376*t3];
         xmm1 = xmm1 ⊙ xmm0;
      }
      M[23371872+8536*t1+8*t2] = xmm1;
   }
```

Figure 2: Matrix multiply as it is extracted from the binary code

instructions from the original program are still necessary: these instructions are then "outlined" and replaced with abstract statements that hide all the architecture-specific intrinsic computations. What remains is a program using and defining scalars registers and one single array representing memory accessed through affine functions. And that is enough for parallelization.

3. PARALLELIZING THE C CODE

After the code extraction described in the previous section, we obtain a C code made of all the memory accesses performed in the binary code but where the semantic is hidden. We do not want to restore the code semantic here in order to keep the code as simple as possible for polyhedral tools later transforming this code. Figure 2 presents the C code as extracted from a binary matrix multiply. The computation is made in register `xmm1` before being written back to memory, represented by the `M` array. Note that the operations on values have all been replaced by a generic operator $\odot$, this operator can be implemented by any operator in the C code used by the transforming compiler, `+` for example. As is, this code fits the polytope model requirements. However, most modern polyhedral compilers fail to take such codes into account, for two main reasons. First, the large values used as coefficients in linear functions often lead to internal errors. Those coefficients are actually due to the linearization of the access functions and to our representation of the memory as an array whose base is at address zero. Splitting the memory into arrays solves this problem. Second, the scalar values `xmm0` and `xmm1`, which are only temporary variables, add some extra dependencies which prohibit many transformations in the polytope model. Those scalar references can be removed in some specific cases.

3.1 Splitting the memory

To help the parallelizer, we simplify memory accesses by splitting non-intersecting memory areas into different arrays and by rebuilding multi-dimensional arrays whenever it is possible. If the loop bounds are non-parametric, those dimensions can be built back by checking all the possible array shapes. In the case of parametric loop bounds, rebuilding the arrays becomes complex and can be solved for some, hopefully frequent, parameter values in association with a runtime check.

```
for (t1 = 0; -1023 + t1 <= 0; t1++)
   for (t2 = 0; -1023 + t2 <= 0; t2++) {
      A2[t1][8*t2] = 0;
      xmm1 = 0;
      for (t3 = 0; -1023 + t3 <= 0; t3++)
         xmm1 = xmm1 ⊙ (A1[t1][8*t3] ⊙ A3[t3][8*t2]);
      A2[t1][8*t2] = xmm1;
   }
```

Figure 3: Matrix multiply after forward substitution

3.2 Removing scalar references

A reference to a scalar variable in a polyhedral loop kernel can have a major impact on data dependencies but existing parallelization tools are currently poorly simplifying those scalar references: they actually expect those simplifications to have already been performed. However, in some cases, removing scalar references is not an easy task.

The first technique which can be applied is forward substitution. One can see on Figure 3 the matrix multiply code after forward substitution. Even if references to `xmm0` have been suppressed, some references to `xmm1` remain. Some compilers implement privatization to solve those remaining cases but privatization seems difficult to implement in the polytope model and this support appeared to be suboptimal in PLuTo [2, 6] and PoCC [8, 9] during our tests. Another commonly suggested transformation is to use results about scalar expansions, contraction and renaming [4, 11]. However, no available tool is currently able to perform simultaneously transformation and memory space optimization efficiently. Moreover, guiding the transformations according to the memory space used by expanded variables is also a complex task. To illustrate it, we present in Figure 4 the matrix multiply code, transformed by PLuTo after expanding `xmm1`. One can see that no contraction is possible anymore with the transformation chosen by the compiler, leading to memory space and performance penalties.

Another solution would be to perform some pattern matching on some specific data flows. In the example from Figure 3, we could identify that, before and after the innermost loop, `xmm1` and `A2[t1][8*t2]` contain the same value. Considering that the array `A2` is not accessed in this innermost loop, references to `xmm1` could be replaced by references to `A2[t1][8*t2]`. This strategy would be easy to implement and would probably lead to decent results. However, we could not guarantee that it could remove every scalar reference. Applying this strategy could lead to a code equivalent to the classical matrix multiply as presented in Figure 5.

Notice that those scalar variables can be suppressed only if we can determine which value they hold after the loop nest. In our example, we may set the scalar variable `xmm0` to `A1[1023][8*1023] * A3[1023][8*1023]` and `xmm1` to `A2[1023][8*1023]`.

```
#pragma omp parallel for ...
for (t1 = 0; t1 <= 1023; t1++)
  for (t2 = 0; t2 <= 1023; t2++)
    xmm1[t1][t2] = 0;

#pragma omp parallel for ...
for (t1 = 0; t1 <= 1023; t1++)
  for (t2 = 0; t2 <= 1023; t2++)
    for (t3 = 0; t3 <= 1023; t3++)
      xmm1[t1][t2] = xmm1[t1][t2]
            ⊙ (A1[t1][8*t3] ⊙ A3[t3][8*t2]);

#pragma omp parallel for ...
for (t1 = 0; t1 <= 1023; t1++)
  for (t2 = 0; t2 <= 1023; t2++) {
    A2[t1][8*t2] = 0;;
    A2[t1][8*t2] = xmm1[t1][t2];
  }
```

Figure 4: Matrix multiply after expansion and transformation

```
for (t1 = 0; -1023 + t1 <= 0; t1++)
   for (t2 = 0; -1023 + t2 <= 0; t2++) {
      A2[t1][8*t2] = 0;
      for (t3 = 0; -1023 + t3 <= 0; t3++)
         A2[t1][8*t2] = A2[t1][8*t2]
               ⊙ (A1[t1][8*t3] ⊙ A3[t3][8*t2]);
   }
```

Figure 5: Matrix multiply after scalar removal.

3.3 Transformations and parallelization

The resulting C code can be parallelized using any source-to-source parallelizing compiler. There is no theoretic restriction on the set of transformations which can be performed by the parallelizer. In our implementation we only prohibit the compilers to fuse or split statements in order to ease the next step. Our implementation currently uses PLuTo [2, 6] as a backend parallelizer.

This genericity allows our method to benefit from any future developments in code optimization techniques. Since any source-to-source compiler can be used, it is also useless to re-implement existing techniques specifically in our system.

4. MERGING THE PARALLEL CODE

4.1 Recovering the semantic

The code used as input for the parallelizing backend is made of the actual memory accesses but is semantically wrong. The correct semantics is actually defined by the binary code statements. We identify which statements in the binary code map to the statements in the parallel C code. To achieve this, we use a PLuTo specific behavior: this compiler numbers the statements in order of appearance in the input sequential code. With other compilers, other information can be used like line numbers in the input code, usually maintained for debugging purposes.

Once the mapping between instructions in the binary code and instructions in the transformed parallel code

```
#pragma omp parallel for private(t2,t3,t4,t5)
for (t2=0; t2<=1023/32; t2++)
  for (t3=0; t3<=1023/32; t3++)
    for (t4=32*t2; t4<=min(1023,32*t2+31); t4++)
      for (t5=32*t3; t5<=min(1023,32*t3+31); t5++) {
        void *tmp0 = (void*)(23371872+8536*t4+8*t5);
        asm volatile("movq $0, (%0)":: "r"(tmp0));
      }

#pragma omp parallel for \
    private(t2,t3,t4,t5,xmm0,xmm1)
for (t2=0; t2<=1023/32; t2++)
  for (t3=0; t3<=1023/32; t3++)
    for (t4=32*t2; t4<=min(1023,32*t2+31);t4++)
      for (t5=32*t3;t5<=min(1023,32*t3+31);t5++) {
        double tmp1 = 0.;
        xmm1 = _mm_load_sd(&tmp1);
        for (t7=0; t7<=1023; t7++) {
          xmm0 = _mm_load_sd((double*)
            (6299744+8296*t4+8*t7));
          __m128d tmp2 = _mm_load_sd((double*)
            (14794848+8*t5+8376*t7));
          xmm0 = _mm_mul_sd(xmm0, tmp2);
          xmm1 = _mm_add_sd(xmm1, xmm0);
        }
        _mm_store_sd((double*)
          (23371872+8536*t4+8*t5), xmm1);
      }
```

Figure 6: Matrix multiply after transformation by PLuTo and semantic restoration.

is found, we can replace statements by inline assembly made of the original statements' code. The original assembly code is not directly injected: some registers or memory accesses have been replaced by linear expressions when extracting the code. Those expressions are evaluated in pure C code before being used in assembly instructions. We also replace SIMD instructions by SIMD intrinsics. Another specificity is that the hardware registers are mapped to C variables in the generated code. Thus, inlined assembly never refers to a specific hardware register but to C variables only. This reduces the constraints on the register allocator of the final C compiler and simplifies code generation.

Figure 6 presents the final code after transformation, parallelization, and semantics restoration. In the presented resulting code, we have used PLuTo to perform the polyhedral transformations, while handling scalar values through privatization, and manually ensuring that the result is correct. Notice that SIMD registers used as variables, inline assembly, and SIMD intrinsics have been generated. The parallelization is achieved using simple OpenMP pragmas.

4.2 Re-injecting the new code in the application

To finalize the code generation, those transformed loop nests must now replace their sequential counterparts in the application. The transformed loop nests are compiled as different functions of a dynamic library, which is loaded using OS facilities when the application starts. Just before the main function call, a runtime component, automatically generated by our tool-chain, inserts breakpoints at loop entries in the sequential application. When those breakpoints are met, the runtime component redirects the execution flow to the corresponding transformed loop nest. This runtime component also communicates the value of the hardware registers to the new loop nest in order to link the variable representing those hardware registers to their actual values. After the transformed loop nest execution, the runtime component writes the value of hardware registers variables back to the actual registers before redirecting the execution flow to the end of the original loop nest. This mechanism ensures the smooth transition between the original code and the transformed loop nests.

Benchmark	Parallelized	In source	Rate
2mm	7	7	100%
3mm	10	10	100%
atax	2	2	100%
bicg	2	2	100%
correlation	3	5	60%
doitgen	3	3	100%
gemm	4	4	100%
gemver	3	4	75%
gramschmidt	1	2	50%
lu	1	2	50%
Average	3.6	4.1	83.5%

Figure 7: Number of loop nests parallelized by our system compared to the number actually present in the source code.

5. FIRST RESULTS

We have implemented most of the framework described in this paper and present in this section some preliminary results that we measured. The scalar replacement step, occurring after the code extraction from the binary code and before its transformation by a polyhedral compiler, has not yet been implemented. In the presented results, it has been performed by hand. The benchmarks used have been taken from the PolyBench [7] benchmark suite. Codes have been compiled with GCC 4.4.5 using the `-O2` optimization flag on a Linux 2.6.35 system. Those codes have been executed on an Intel Xeon W3520 with four processor cores and two threads per core.

5.1 Code coverage

Our system was able to detect all the loops in the test programs, and to transform a vast majority of them. Some loop nests have not been parallelized by our system because of function calls present in the loop body: we conservatively ignore the loop nest in that case. We plan to implement a mechanism such as inlining or interprocedural analysis to handle those loop nests. Since the called functions are often mathematical functions which do not perform any write operations on memory, we expect good results from such mechanisms. In Figure 7, we present the number of parallelized loop nests

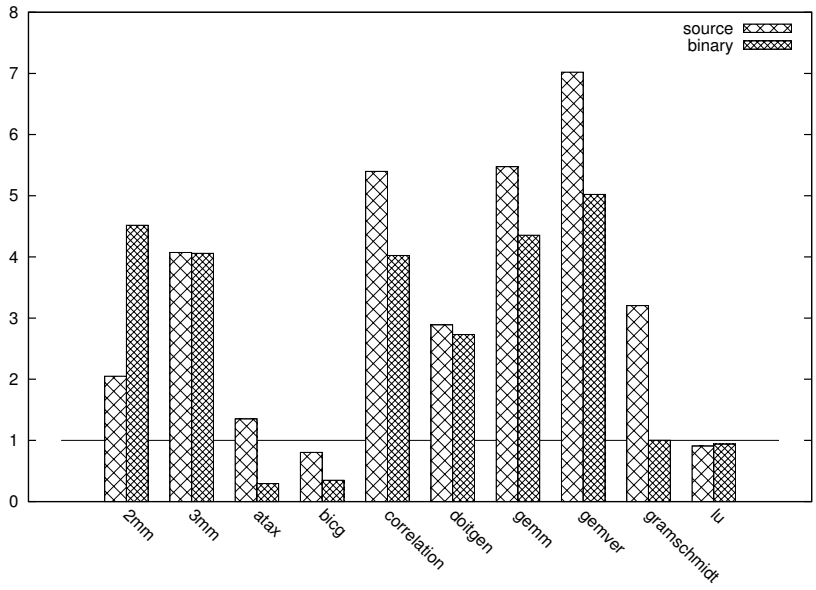

Figure 8: Speedup of each code when parallelized from the source code or by our system.

compared to the number of loop nests in the source code. We can see a current average coverage of 83.5% of the loops actually present in the source code.

5.2 Runtime overhead

Our current implementation uses the Linux `ptrace` mechanism to redirect the execution flow from the original code to the parallel loops. This has obviously an impact on performance, linearly depending on the number of redirections. We measured this overhead to be around a tenth of milliseconds per loop nest execution on our test platform. It means that, in order to obtain speedup, the parallelization must lead to an execution time gain greater than a tenth of milliseconds, which seems quite reasonable.

5.3 Raw performance evaluation

In Figure 8, we present an overview of our system performance. One can see the speedup over the sequential version of each code when parallelized from the source code using PLuTo (denoted `source` in the figure) or when using our system (denoted `binary` in the figure).

We can see that our system can reach speedups comparable with the ones resulting from a source code parallelization. Note that most of the benchmarks in this suite have sequential execution times around a second. This causes the runtime overhead to become quite significant when measuring our system performance. The slight difference between the source code and the extracted code is the cause of the performance gaps as it yields to important differences between the generated parallel schedules.

The `atax` program illustrates the source code sensitivity of PLuTo. Some differences appear in the extracted code compared to the original source code. For example, some parameters in the source code become constant values in the binary codes. Such differences lead to different transformations, and different execution times. In the case of `atax`, this performance gap is favorable to the source code version.

With `2mm`, one can notice that our system significantly outperform the source code parallelization. This is due to an error induced by the source code parallelization: if the tiling is activated in PLuTo, the program does not terminate normally. We then deactivated tiling when parallelizing the source code of this particular program, leading to a poor performance of the source code parallelization.

One can also observe that our system fails to generate an efficient parallelization of `gramschmidt`. In this particular program, the main computation loop nest contains a call to the mathematical library, currently prohibiting the transformation of this loop.

6. RELATED WORKS

Polyhedral loop transformation is now a well established theory and many tools exist to perform parallelization and transformations in this model such as PLuTo [2, 6] or PoCC/LetSee [8, 9]. Those tools provide parallelization from source to source, usually C or FORTRAN programs can be handled.

Many tools like PIN [5] or DIABLO [12] provide some facilities to analyze, transform, and, for some of them, create a new transformed application. Our work can be considered as a specific usage of such frameworks and could have been implemented using anyone of them.

To our knowledge, only two recent papers present a solution to binary code parallelization. First Yardımcı and Franz have proposed a dynamic system to vectorize and parallelize binary codes [13]. They are focused on loops or recursive functions with no dependence, which limits the scope of their system. They do not perform any loop transformation and the dynamic approach is not well suited to heavy transformations as polyhedral ones which require long compilation times.

Second, Kotha *et al.* proposed a framework [3] similar to the one described in this paper with a few significant differences. First, the analysis of the binary application they perform can only find loops and memory accesses that follow a restrictive pattern, reducing the scope of loops that can be parallelized. They decide whether to parallelize or not using non-exact dependence testing and do not perform any loop transformation. Our analysis does not have those restrictions: we perform a state of the art dependence analysis, we are able to perform loop transformations, and our use of C code as intermediate representation allows us to avoid re-implementing existing compilation techniques.

7. PERSPECTIVES

Using a polyhedral parallelizer in our system has revealed some weaknesses of polyhedral tools in supporting scalar variables. Currently, those tools expect codes where the temporary scalars have been deleted. However this removal is not always straightforward. We plan to implement an automatic scalar removal pass in our tool in order to help the parallelizing compilers.

As our intermediate representation is a valid C program, any source-to-source optimizer can be used, making it possible to extend advances in compiler construction to binary applications with nearly no implementation cost. Among all the possible backends, one could imagine to apply vectorization or speculative parallelism for example.

In the current implementation, the extracted C code is focused on memory accesses for data dependence analysis. Nothing prohibits the construction of C code focused on other features like control flow for example, allowing backends to target other kinds of optimizations.

8. CONCLUSION

We propose a framework able to transform and parallelize binary codes using polyhedral transformations. Partially raising the binary code to C code provides the opportunity of using any source-to-source tool to transform or parallelize code with nearly no implementation cost. First results show that speedups similar to those obtained with source parallelization can be expected. Its efficient analysis process enables to detect and handle most of the loops present in binary codes, despite the optimizations performed by the original compiler.

9. REFERENCES

[1] The LLVM compiler infrastructure. http://llvm.org.

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

[3] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua. Automatic parallelization in a binary rewriter. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 547 –557, 2010.

[4] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Comput.*, 24:649–671, 1998.

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, pages 190–200, 2005.

[6] PLuTo website. http://pluto-compiler.sourceforge.net/.

[7] PolyBench website. http://www-roc.inria.fr/~pouchet/software/polybench.

[8] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI'08*, pages 90–100. ACM Press, 2008.

[9] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *CGO'07*, pages 144–156. IEEE Computer Society press, 2007.

[10] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using dj graphs. *ACM Trans. Program. Lang. Syst.*, 18:649–658, 1996.

[11] W. Thies, F. Vivien, and S. P. Amarasinghe. A step towards unifying schedule and storage optimization. *ACM Trans. Program. Lang. Syst.*, 29(6), 2007.

[12] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology*, pages 7–12. IEEE, 2005.

[13] E. Yardımcı and M. Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 127–138. ACM, 2006.