ABSTRACT

Storage mapping optimization is a flexible approach to folding array dimensions in numerical codes. It is designed to reduce the memory footprint after a wide spectrum of loop transformations, whether based on uniform dependence vectors or more expressive polyhedral abstractions. Conversely, few loop transformations have been proposed to facilitate register promotion, namely loop fusion, unroll-and-jam or tiling. Building on array data-flow analysis and expansion, we extend storage mapping optimization to improve opportunities for register promotion.

Our work is motivated by the empirical study of a computational biology benchmark, the approximate string matching algorithm BPR from NR-grep, on a wide issue micro-architecture. Our experiments confirm the major benefit of register tiling (even on non-numerical benchmarks) but also shed the light on two novel issues: prior array expansion may be necessary to enable loop transformations that finally authorize profitable register promotion, and more advanced scheduling techniques (beyond tiling and unroll-and-jam) may significantly improve performance in fine-tuning register usage and instruction-level parallelism.

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

General Terms
Algorithms, Performance.

Keywords
Register Promotion, Tiling, Blocking, Scheduling, Array Contraction, Array Folding, Pattern Matching, String Matching, Itanium.

1. INTRODUCTION

Scalar optimizations applied to register promotion are ubiquitous in production compilers and successful in improving performance of a large spectrum of programs. Most of them are based on constant sub-expression elimination or partial redundancy elimination, combined with pointer analysis and loop unrolling [34, 31, 7]. Register promotion for large register files must be combined with more aggressive loop transformations. First of all, it is critical to exploit scalar reuse over iterations of the innermost loop [16], e.g., to optimize the benefits of software pipelining. Register tiling is a complementary approach to exhibit scalar reuse at different depths in a loop nest; it plays the first role in optimization strategies for many memory-bound kernels from numerical, multimedia, cryptography, or computational biology benchmarks. A typical example is matrix-matrix multiplication [36], it is also the case for the computational biology example described in this paper. Most works build on locality-improving techniques like loop fusion [8, 10, 33] or tiling [11, 33, 21] to exhibit more opportunities for scalar promotion. In this paper, we will refer to both tiling and unroll-and-jam based techniques as register blocking. Recent improvements to these approaches proved their applicability to wide issue machines [21] and codesign and synthesis tools [42, 43].

1.1 Running Example

We will now present a simple example to clarify the basic register blocking concepts and to motivate the use of array expansion and storage mapping optimization. The nest in Figure 1 is typical of hand-optimized programs with scalar reuse, both intra-loop — n and p — and loop-carried — n and o. Dependences on D (and scalars) hamper register blocking: it does not seem possible to further reduce the number of memory accesses.

Dependence removal

To enable unroll-and-jam or tiling, one may first convert D to the single-assignment array E in Figure 2, thanks to array data-flow analysis [17], then eliminate scalars n, o and p by forward substitution, see Figure 3. The expansion of D removes all memory-based dependences on arrays, while forward substitution removes output and anti dependences on scalars n and o (at the cost of redundant memory accesses, but at a lower cost than expanding scalars n and o into arrays). Notice other dependence removal techniques are not really applicable to this case: array renaming is not applicable [24], and array privatization [46] does not easily apply in the presence of loop-carried dependences and induces a copy overhead that may only be fully eliminated by array data-flow analysis, the core technique for conversion to single assignment form. One may now unroll the outer loop, fuse the resulting inner loops, and finally unroll the inner loop, see Figures 4 and 5 (we use unroll factors of 2 and assume m and k are odd numbers for the sake of clarity). Figure 7 shows array data-flow dependences (arrows) and the resulting load (L) and store (S) pattern on the 2 × 2 (fully unrolled) tile. A simple scalar promotion algorithm [34] will only take intra-loop reuse into account and lead to the code in Figure 6. The corresponding load/store pattern is shown in Figure 8.
the hand-optimized version dedicated
the lack of scalars reuse along the inner loop (vertical dependence):
the array is now two-dimensional. The additional load comes from
wald et al. [16], but along the
scalar reuse across loop iterations, most notably the one by Duester-
plying more advanced rescheduling transformations before scalar
for register blocking. Of course, this does not improve when ap-
ins the applicability of array expansion as an enabling technique
both leftward and downward tiles. Such a disappointing result ru-
phases
eliminated by array contraction [38], since live values flow from
loaded from
store is traded for an additional load every four iterations of
fully unrolled, and for complex loop bounds and dependence pat-
terns, like the skewed version of our benchmark example in the next
section. In any case, such techniques would not succeed in folding
the expanded arrays.

Exploring a more ambitious approach
We face two challenges:

1. enable loop-carried and cross-loop scalar reuse at every level
   of an imperfect loop nest, in the context of complex (but reg-
ular) array data-flow information:

2. guarantee that the memory footprint will be minimal, undo-
   any unnecessary array expansion induced by the conver-
sion to single-assignment form.

To address the first issue, we duplicate the declaration of array E
for each surrounding loop. The new arrays are called reuse buffers:
they store every value that will be reused across the corresponding
loop iterations. Practically, right after unroll-and-jam, we duplicate
writes to E[i+1][*] — values flowing to the next iteration of
the outer loop — with references to a new array B, and we replace reads
from E[i-1][*] — values flowing from the previous iteration of
the outer loop — by references to B, see Figure 9. After unrolling
the inner loop, we repeat the process and store the values reused
along the inner loop in a new array C, see Figure 10. This technique
generalizes the inter-iteration reuse mechanism of [16] and extends
the virtual register concept [31] to arrays. These progresses make
inter-iteration and cross-loop reuse applicable to a large class of
imperfectly nested loops and arbitrary loop levels.

Next, we may reduce the memory footprint with array folding
techniques. The second dimension in array C is easily removed by
array contraction, but arrays E and B require a finer-grain folding

techne. This is precisely what storage mapping optimization is
intended for [26]. Studying the number of intermediate writes
during the lifetime of values produce at every iteration, one may

```
for (i=0; i<m; i++)
    o = n = D[0];
for (j=0; j<k; j++)
    p = D[j];
    n = f(n, o, p);
D[j] = n;
```
check that only two columns of B (separated by a useless column recording no value) and two lines and two columns of E are simultaneously alive. Analogously, one may improve the folding of array C since only two of its elements are simultaneously alive. Using the technique by Lefebvre and Feautrier, we obtain the code shown in Figure 11 (the integer division in the subscript of E is due to the removal of the useless column). \(^1\) The corresponding load/store pattern is shown in Figure 13.

The introduction of buffers B and C split the iteration space of values produced and consumed within a tile — array E — from the iteration space of values flowing to the upward tile — array C — and from the iteration space of values flowing to rightward tiles — array B. Storage mapping optimization exploits this iteration-space partitioning to decouple the folding of each array, B, C and E, hence to further reduce the memory footprint, and to discover opportunities for scalar promotion: it is now straightforward to promote the small, bounded size arrays C and E to scalars. Applying intra-block scalar promotion to B as well, we obtain the optimized code in Figure 12. The corresponding load/store pattern is shown in Figure 14.

This final version needs only three fourths of the loads and half the stores of the original code. In addition, considering the peeled iteration \(j = 0\), scalar promotion eliminates all array references but one, reusing values of C when entering the inner loop and reusing values of B across iterations of the outer loop. This kind of scalar reuse is out of reach of previously proposed techniques.

1.2 Facilitating Register Promotion

Previous works studied loop transformations facilitating register promotion (mostly) in isolation from the other optimization phases. A real loop nest optimizer includes transformations to deal with the cache hierarchy and to exploit instruction or thread level parallelism. Unfortunately, controlling the interplay of loop transformations is one of the hardest problem for today's loop-restructuring compilers: first of all, predicting the effect of loop transformations is tough on micro-architectures with complex dynamic structures like branch predictors and reorder buffers, and predicting their interactions is even worse; in addition, classical transformations do not compose easily because they rely on fragile pattern matching rules.

Of course, no technique is supposed to solve all performance problems, but a practical loop transformation for register promotion should minimize the constraints imposed on other transformation phases. For example, loop distribution or skewing [1] may favor software pipelining, exposing more ILP without degrading instruction cache locality. Register promotion should not forcibly

---

\(^1\)Their method generates modulo operations for arrays C and E as well, of the form \(C[i%2]\) and \(E[i%2]\), but these subscripts can be further simplified because \(i\) and \(j\) are odd numbers. Likewise, notice that the complex subscript of array B may be simplified by further unrolling (or strip-mining) the outer loop.
tile loops and rule out other important transformations. Our work
addresses this interplay issue by three means:

- decoupling the array folding technique from the scheduling
  algorithm or profitability heuristic for loop transformation;
- decoupling the array folding technique from unrolling and
  low-level register handling mechanisms (e.g., rotation);
- relying on storage-mapping optimization [26, 44, 40, 45], a
generalized approach to array folding compatible with the
most advanced scheduling algorithms.

We describe a polyhedral approach combining array expansion
with affine scheduling and storage mapping optimization.

Array expansion

Unlike other array expansion techniques, conversion to (dynamic)
single-assignment form [17] removes every memory-based (output
and anti) dependence. This improves the potential for loop trans-
formations and reduces the impact of syntactic variations in the
indexing scheme of the original arrays. Maximal static expansion
[2] is a natural extension suitable for irregular nests: it removes a
memory dependence only when the producer of a value is statically
known (i.e., without evaluating & functions at run-time).

At first glance, array expansion seems the exact opposite of our
final goal. However, removing spurious dependence constraints fa-
vors the combination of register promotion with other optimiza-
tions. Interestingly, it also favors register promotion itself by en-
abling alternative array folding opportunities with better impact on
memory traffic. In addition, we will show that storage mapping
optimization controls the overhead of array expansion on the re-
main ing memory accesses.

Affine scheduling

Most schedule-oriented loop optimizations can be modeled with
affine schedules. It is the case for unimodular transformations, loop
fusion, unroll-and-jam, tiling, software pipelining and statement re-
ordering [22, 27, 5], as well as more complex transformations like
shackling [25] or chunking [6]. In addition, several recent locality-
improving heuristics build directly on affine schedules [45, 29, 6].
The abstraction level of this model allows for register reuse across
iterations of any loop and across different loops, without suffering
from nesting restrictions.

Storage mapping optimization

Designed to reduce the memory footprint after a wide spectrum
of loop transformations, storage mapping optimization is a flexible
generalization of array contraction [28, 38]. It supports both uni-
form dependence vectors [9, 44] and polyhedral abstractions [26,
40, 45]. The last three techniques benefit from array data-flow anal-
ysis [17, 39, 47, 3] and affine scheduling [19] to discover and
exploit liveness information.

Our work extends the algorithm by Lefebvre and Feautrier [26].
It can be generalized to irregular loop nests along the lines of [13],
and this is the main reason why we prefer this technique to the
algorithm by Quilleré and Rajopadhye [40]. However, we would
not be able to directly apply the techniques by Thies et al. [45]
because it cannot contract more than one array dimension, which
is required to reduce memory footprint after conversion to single-
assignment form.

Eventually, whereas traditional storage mapping folds multiple
array elements to a single location to reduce the memory footprint,
our technique combines this effect with buffer insertion (or index-
set splitting) to decrease the memory access rate in a loop nest, and
to factor memory transfers to folded locations.

1.3 Applications and Contributions

Our work is motivated by the empirical study of a computational
biology benchmark — the approximate pattern matching algorithm
BPR from NR-grep [35] — on a wide issue architecture. Our ex-
periments confirm the major benefit of register blocking (even on
non-numerical benchmarks) but also shed the light on two novel
issues: prior array expansion may enable loop transformations that
authorize profitable register promotion, and combination with alter-
native scheduling techniques (beyond interchange and fusion) may
significantly improve performance in fine-tuning register usage and
instruction-level parallelism.

Our main contributions are the following.

Register promotion. We revisit loop transformations for register
promotion in a more general setting, building on the most ad-
vanced techniques in array data-flow analysis, array expan-
sion, storage mapping optimization, and affine scheduling.

Storage mapping optimization. We extend a folding technique to
better handle tiled iteration spaces and exploit the topmost
level of the memory hierarchy. The main idea consists in
capturing inter-iteration and cross-loop reuse patterns into
so-called reuse buffers.

Computational biology. We present an optimized implementation
of an approximate pattern matching algorithm for modern
microprocessors. We experience strong speed-ups — over
6 on the Itanium1 and 5 on the Itanium2, with a number of
instructions per cycle (IPC) close to optimal. This promises a
significant impact on the relative merits and applicability
ranges of pattern matching algorithms.

The paper is organized as follows: Section 2 discusses the op-
timization of a computational biology benchmark and further mo-
tivates our approach, Section 3 presents our algorithm for register
promotion in the polytope model, before the conclusion.

2. BENCHMARK APPLICATION

We study the pattern matching algorithm BPR of practical use
for computational biology (and some data mining applications). It
is one of the many approximate string matching algorithms imple-
mented in the NR-grep utility (non-deterministic reverse grep ver-
sion 1.1.1) [35].

As opposed to well known tools like grep and agrep, NR-grep is
based on the bit-parallel simulation of a non-deterministic au-
tomaton. Practically, it makes use of each individual bit in a 64 bit
register to encode up to 64 states simultaneously. In addition, NR-
grep implements approximate string matching algorithms, suitable
for computational biology.

Practically, the implementation of BPR takes a (long) text and a
length m pattern, and search for occurrences of the pattern with at
most k errors (character insertion, deletion, substitution or swap).
Useful values of k range from 0 (exact string matching) to m/2,
and beyond for large alphabets. The main kernel of BPR is found
in file esimple.c, function esimpleScan in the “forward” case
with k ≥ 3, see Figure 15.

Compared to the running example, BPR is not perfectly nested,
uses one additional array T with similar access patterns, and imple-
ments a lot of bitwise logical operations. Interestingly, it also in-
cludes subscript of subscript references (to a read-only array) and
uses irregular control structures to report matches (unpredictable
early exit). None of these irregularities is a serious threat for array
and loop transformations, but both of them suggest the applica-
tion of modern techniques like fuzzy array data-flow analysis [14,
ILP would also be impossible. The only way to apply the required transformations is to resort to array expansion (with array data-flow analysis), relying on a phase of storage mapping optimization to reduce the memory footprint and provide opportunities for register promotion.

### 2.2 Experiments

We used an Itanium2 1.3 GHz (Madison) workstation with HP's ZX1 chip-set and Intel's Electron compiler version 8. Measurements were conducted for several values of the pattern length $m$, alphabet size $\sigma$, and error $k$. We checked that the number of matches (early exists) never exceeded a few thousands.

First of all, it appears that skew and pllm — the second and third candidate version in Figure 16 — have very similar performances; we will thus limit ourselves to the results on rect and skew.

We consider two data-sets:

- $dna$ corresponds to searches in the 600 kilobase genome of the *buchnera* bacteria ($\sigma = 4$), with pattern size $m = 41$;
- $txt$ corresponds to searches in a 10 megabyte English $latex\$\texttt{"{L}atex}$ document ($\sigma = 127$), with pattern size $m = 24$.

Speed-ups are measured with respect to the base version of NRgrep optimized with Electron (best optimization parameters with profiling), running for approximately 150 million cycles on $dna$ (more than 100 milliseconds) and 1500 million cycles on $txt$ (more than 1 second).

The tile's width and height are denoted by $p$ and $q$, respectively. We first study the combined effect of all parameters except $q$.

Figure 17 shows the speed-ups of the two optimized versions (rect and skew), on the two data-sets ($dna$ and $txt$), for $k = 13$ and $k = 19$ errors, for tile widths $p = 8$ and $p = 16$, and for the best tile height. Performance of both versions is very good in all cases — speed-up of 3 or more. Yet we experience significant variations, depending on the version, data-set and parameters.

- The schedule of the nest is important: the skewed and tiled version is always better than tiling alone, up to 25.8% for $dna$, $k = 19$ and $p = 16$. This is due to a better exploitation of ILP without deteriorating locality.
- Increasing the tile width has a rather unpredictable impact. This is due to the large size of the loop body after full unrolling of the inner loops. Inspecting the generated code shows that the compiler sometimes produce spill code, having exhausted the 128 available registers.

<table>
<thead>
<tr>
<th></th>
<th>rect</th>
<th>skew</th>
<th>rect</th>
<th>skew</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dna$</td>
<td>$k = 13$</td>
<td>$k = 13$</td>
<td>$k = 19$</td>
<td>$k = 19$</td>
</tr>
<tr>
<td>$p = 8$</td>
<td>2.99</td>
<td>3.31</td>
<td>4.60</td>
<td>5.08</td>
</tr>
<tr>
<td>$p = 16$</td>
<td>2.98</td>
<td>3.45</td>
<td>4.06</td>
<td>5.11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>rect</th>
<th>skew</th>
<th>rect</th>
<th>skew</th>
</tr>
</thead>
<tbody>
<tr>
<td>$txt$</td>
<td>$k = 13$</td>
<td>$k = 13$</td>
<td>$k = 19$</td>
<td>$k = 19$</td>
</tr>
<tr>
<td>$p = 8$</td>
<td>3.00</td>
<td>3.40</td>
<td>3.98</td>
<td>4.28</td>
</tr>
<tr>
<td>$p = 16$</td>
<td>2.99</td>
<td>3.54</td>
<td>3.41</td>
<td>4.05</td>
</tr>
</tbody>
</table>

**Figure 17:** Best speed-ups for $p = 8$ and $p = 16$.

These results are confirmed by the IPC metric. On most combinations of parameters and versions, we get over 5 IPC, with a peak 5.6 on the best case, for an optimal value of 6 on the Itanium. However, for some cases, performance is sub-optimal; e.g., 3.79 IPC for $k = 19$, $p = 16$, version rect and data-set $txt$. 

---

**Figure 15:** Kernel of the BPR algorithm
Let us now study the effects of the tile height \( q \). Figure 18 shows the speed-up for \( \alpha_n \), \( k = 19 \), \( p = 8 \) and \( p = 16 \), varying the tile size \( q \) from 3 to 10. Beyond the performance advantage of \textsc{skew} with respect to \textsc{rect}, the results are hard to predict from a compiler perspective. Even this advantage comes down to a tiny difference for some cases, like \( p = 8 \) and \( q = 5 \). Most of these variations are due to the complex interplay of the tile shape with the relative size of the prelude/postlude of the tiled loops (when \( q \) does not divide \( k + 1 \)), and with the back-end optimizer of the Electron compiler (software-pipelining, register allocation, spilling heuristic).

These results confirm that predicting the interplay of loop transformations is a tough problem for modern compilers and architectures. We thus advocate for a flexible approach to register promotion that does not restrict the applicable loop transformations to register blocking alone. Any other framework would not be applicable in future optimizers that are likely to rely on feedback information and iterative (or adaptive) schemes [23, 15, 20].

To conclude this section, we showed that both array expansion and storage mapping optimization are necessary to optimize the memory-bound BPR kernel. Interestingly, we also showed that significant performance benefits may be obtained by combining this extended register blocking technique with other performance-increasing transformations, like skewing to increase ILP. The next section will take the point of view of compiler designers. It will describe how this combination of transformations is made possible by the expressiveness of multidimensional affine schedules [19].

### 3. POLYHEDRAL TRANSFORMATIONS

Our application to the BPR algorithm showed that dependence removal is not only important for parallelization (ILP in this case), but for register promotion as well. It enables locality-improving transformations and leads to strong speed-ups.

#### 3.1 State-of-the-Art Array Folding

Let us summarize the storage mapping optimization method by Lefebvre and Feautrier [26]. In this presentation, we assume a static-control nest [17], i.e., affine bounds, conditional guards and subscripts with respect to surrounding loop counters and symbolic constants, but our method extends to irregular nests through the extended algorithms in [3, 13].

**About the polytope model**

We first recall classical results and definitions. Each iteration of a statement is called an instance. The instance of a statement \( s \) for an iteration vector \( v \) (the vector formed by the surrounding loop counters) is denoted by \( (s, v) \).

The polytope model assume that the relative ordering of every iteration of every statement is fully characterized through a collection of affine schedules. An affine schedule is a function from iteration vectors to lexicographically ordered time vectors. For a given statement \( s \), the corresponding function \( \theta_s \) is called the affine schedule of \( s \) [19]. This model can represent parallel and sequential schedules obtained through most loop transformations [22, 5].

Array data-flow analysis yields, for each reference \( r \) in right-hand side of a statement \( s \), the function \( \text{RD}_{s,r} \) mapping any iteration of \( s \) to the precise instance that produced the value read through \( r \) (using the PIP tool [18]); it is a generalization of the classical reaching definitions [34] to polyhedral sets of instances and arrays. When a reference reads a value defined before the program fragment of interest, array data-flow analysis yields the special \( \perp \) instance. In general, \( \text{RD}_{s,r} \) is a union of polyhedra and can be represented by a quasi, short for quasi-affine selection tree [17] (a generalization of a last-write tree [32]). Each leaf in a quasi is an affine function — characterized as a convex polyhedron — mapping a disjoint set of instances of \( s \) to their definitions. For example, if \( s \) is the second assignment in the running example, \( p = \text{D}[j] \), and \( t \) is the fourth assignment, \( \text{D}[j] = \alpha \), then

$$\text{RD}_{s,r}(s,i,j) = \{ \begin{array}{l l} \perp & \text{if } i = 0 \\ (t, i-1, j) & \text{else} \end{array} \}$$

Indeed, reference \( \text{D}[j] \) in \( s \) reads the value produced by the last iteration of \( t \), except for the first iteration of the outer loop where it reads the initial value of array \( \text{D} \).

Array expansion is a direct application of array data-flow analysis [17]. Back to the running example, one may convert array \( \text{D} \) to the single-assignment array \( \text{E} \) in two simple steps:

- replace every reference to \( \text{D} \) in left-hand side by a reference to \( \text{E} \) subscripted with the surrounding loop indices, statement \( t \) becomes \( E[i][j] = \alpha \);
- replace every reference to \( \text{D} \) in right-hand side by a \( \text{C} \) implementation of the quasi of its reaching definition, where \( \perp \) corresponds to the original references to incoming values of \( \text{D} \), and where non-\( \perp \) leaves yield subscripts of \( \text{E} \), statement \( s \) becomes \( p = \text{RD}_{s,r}(s,i,j) \), i.e.,

$$p = (s=0) ? \text{D}[0] : E[i-1][j].$$

This naive implementation of the right-hand side can be further optimized by polyhedral code generation techniques [41], hoisting all conditionals out of the inner loops. Modern code generation tools generate excellent control structures with no or very low overhead on medium-sized loop nests; for example, Bastoul’s code generator [4] generate nearly-optimal conditionals and loops for hundreds of

---

**Figure 16: Three candidate schedules for the optimized BPR kernel**
From the lexicographic order, iteration (\(i,j\)) of statement 1 executes before iteration (\(i,j+1\)) of statement 2, which itself executes before iteration (\(i,j+1\)) of statement 1, etc. Thanks to the flexibility of affine schedules, we will show that storage mapping optimization smoothly handles inter-iteration and cross-loop reuse.

For each depth \(k\) of the loop nest, one may deduce a function \(w^k_s, q\) mapping each instance \(i\) of a statement \(s\) to the set of inter-fering writes following \(i\) according to the affine schedule (using the PolyLib) — this set is called the working set of \(i\). One may then compute the maximal reuse distance \(m^k_{s,q}\) between each instance of \(s\) and its last interfering write at depth \(k\) (using PIP). It was proven in [26] that a single-assignment array subscript \([i_1]\ldots[i_d]\) in left-hand-side of \(s\) may be safely replaced by

\[
[i_1](1 + m^k_{s,q})\ldots[i_d](1 + m^k_{s,q})
\]

The algorithm is formally stated in Figure 20. It uses the following additional definitions: \(d\) is the depth of the full nest and \(\text{INTERSECT}\) is the intersection of polyhedra. Both \(\text{INTERSECT}\) and the computation of \(\Delta^k_{s,q}\) correspond to PolyLib functions.

In a final step, each array in left-hand side of a statement is given a unique name, and the effects of these transformations are propagated to the uses in right-hand side. In addition, non-convex array access patterns (lattice footprints) with strided rows or columns of useless data (never read nor written) can be further contracted by dividing the corresponding subscripts by the strides' greatest common divisor [40]. We applied optimization to array \(b\) in Figure 11.

**Application to the running example**

Let us now informally apply this algorithm to the code in Figure 5. The first statement stores a value in \(E[i_1][j]\) that does not escape the loop body. The second and fourth statements record in \(E[i_1+1][j]\) and \(E[i_1+1][j+1]\) a value that is later consumed by
the next iteration of the outer loop. The third statement records in \( E[1][3+1] \) a value that is later consumed by the next iteration of the inner loop. No value flows across more than one iteration of the outer loop: the maximal distance at depth 1 is 1. With a fixed value of 1, some values (produced by the second and fourth statement) flow across all iterations the inner loop: the maximal distance at depth 2 is 2, hence \( m_2^1 = m_2^2 = m_2^3 = \infty \), i.e., there is no folding opportunity. One may thus fold array \( E \) down to two columns: \( E[1][3] \) can be replaced by \( E[1][4][2][3] \), which is equivalent to \( E[1][5] \) since 1 is an odd number. No renaming is necessary here since \( E \) is a temporary single-assignment array.

We have shown that array contraction cannot remove any dimension after conversion to single-assignment form and unroll-and-jam. Storage mapping optimization does a better job: the folded array is only twice the size of array \( E \) in the original nest.

**Important remarks**

Interestingly, the **number of folded dimensions** of the resulting arrays is proven maximal for all affine schedules and storage mappings based on dimension-per-dimension foldings [40]. This strong result provides a "no-harm" guarantee for conversion to single-assignment form: if no schedule transformation is applied (fusion, tiling, or folding) but does not support any folding of the first dimension of the result provides a "no-harm" guarantee for conversion to single-assignment form: if no schedule transformation is applied (fusion, tiling, or folding), then it is easy to propagate the new buffer in all references accessing values reused over iterations at depth \( k \). This process may be repeated for all statements (or only those of interest to inter-iteration scalar promotion). In the resulting nest, we have transferred to the separate new buffers, all references reading values produced at a different iteration of a surrounding loop at depth \( k \).

Applying this technique to the outer and inner loops of the running example — respectively at depth 1 and 2 — produces the code in Figures 9 and 10, introducing the new arrays \( B \) and \( C \).

Notice these reuse buffers are just a convenient way to implement index-set splitting: they easily separate iterations producing values that "escape" to other iterations, from the ones producing local values only.

**Folding heuristic**

Despite the optimality result and alternative strategies proposed in [40], there is no reason for choosing a computation order or another for the maximal reuse distances at depth \( k (m_2^k) \). Both [26] and [40] make arbitrary choices, like folding array dimensions from the outermost loop inwards.

In the context of register promotion and reuse buffers, the situation is different. Recalling the **hierarchical boundary** concept introduced in this section, we may safely begin the computation of \( m_2^k \) from the inner loops, taking the risk of generating a little more variable names in the innermost loop body. This is very acceptable, assuming that a good register allocation algorithm is implemented in our compiler back-end (to carefully trade register pressure for ILP). The evaluation of \( m_2^k \) for the outer loops may lead to smaller values since the working sets have been restricted to fixed values of the inner loop counters. Practically, we thus reverse the ordering proposed by [26]: we compute \( m_2^k \) from the innermost level outwards.

This optimization is responsible for the reduction of the memory footprint in the \( \text{gllm} \) version of BPR. The reuse buffer (across the outer loop) has half the storage size of the two other versions, i.e., exactly the same size as arrays \( D \) and \( T \). This is due to subtle interactions between the affine schedule and the liveness of values stored in the reuse buffer.

**Algorithm summary**

It is not the purpose of this paper to discuss locality-improving schedule transformations. We will thus assume that an arbitrary scheduling algorithm is applied to the single-assignment loop nest. In the case of the running example, we would reproduce the manual transformations through two-dimensional tiling. For the application to NR-grep, this scheduling algorithm would optionally combine two-dimensional tiling with a skewing transformation, either before or after tiling, to reproduce the three candidate versions presented in Section 2.1.

For simplicity reasons, it is much better not to unroll the loops until the final scalar promotion phase. Indeed, storage mapping automatically insert a buffer to decouple inter-iteration reuse from intra-iteration reuse at depth \( k \). If the left-hand side of \( s \) is of the form \( A[1][1] \ldots [1][d] \), and \( if \ and \ only \ if \) some values defined by \( s \) flow to other iterations of the surrounding loops, we insert a new statement \( s' \) right before/after \( s \) replicating the stored value in a new buffer \( \text{Neuter} \). Practically, we create a statement \( s' : \ \text{Abuffer}[1][1][d] = A[1][1][d] \), with the same schedule as \( s \) and whose iteration domain is restricted to the set of iteration vectors \( v \) such that there exists an iteration vector \( v' \) of a statement \( s \) with a reference \( v' \) in right-hand side such that \( \text{RD}_{v'}(v') = v \) and \( v < v' \). Thanks to the result of array data-flow analysis again, it is easy to propagate the new buffer in all references accessing values reused over iterations at depth \( k \). This process may be repeated for all statements (or only those of interest to inter-iteration scalar promotion). In the resulting nest, we have transferred to the separate new buffers, all references reading values produced at a different iteration of a surrounding loop at depth \( k \).
optimization is an expensive process (most polyhedral operations are exponential in the worst case). We thus prefer tiling to unroll-and-jam to improve locality.

Putting it altogether, the main steps of our extended algorithm are as follows.

1. Run array data-flow analysis for every right-hand side array reference.
2. Convert all temporary arrays and scalars (not escaping the nest) to single-assignment form. Alternatively, forward substitution may be used to remove memory-based dependences on scalars, reducing the memory footprint of the expanded program. On demand, one may expand some escaping variables as well by insertion of copy-in,copy-out code.
3. Apply any affine scheduling algorithm, typically a locality-improving one like hierarchical tiling. This scheduling phase may include other loop transformations to address performance issues like instruction and thread level parallelism.
4. Fix a hierarchical boundary, identifying the aggressively unrolled inner loops where scalar promotion will occur. This boundary would typically correspond to the tile dimension, i.e., depth 3 in the tiled running example.
5. Insert reuse buffers at all depths below (enclosing) the hierarchical boundary to capture values flowing across iterations of the outermost loops (not only the innermost one).
6. Apply storage mapping optimization from the inner loops outwards.
7. Fully unroll the inner loops, up to the hierarchical boundary.
8. Apply a classical scalar promotion algorithm to eliminate as many array accesses as possible.
9. Apply forward substitution to remove scalar duplicates. This may require further unrolling of the inner loops or make use of architectural features like rotating register files.

4. CONCLUSION AND FUTURE WORKS

We revisited loop transformations for register promotion in the more general setting of polyhedral loop transformations, array expansion and storage mapping optimization. This is motivated by a detailed study of the limitations of classical approaches — including scalar reuse across outer-loop iterations and array folding — and by empirical evidence on a real-world computational biology example. Besides this compilation issue, our strong speed-ups on this kernel modify the fragile balance between the relative merits of pattern matching algorithms [35].

On the algorithmic side, we combine array expansion with other enabling transformations to improve the opportunities for register promotion. We also extend a schedule-dependent storage mapping optimization technique to better handle tiled iteration spaces and explicitly manage inter-iteration reuse through specific buffers. Finally, we sketched a flexible register blocking algorithm that integrates these techniques, without restraining the application of loop transformations targeting other architectural components (caches, predictors, ILP, threads, etc.). We manually applied this algorithm to reproduce the speed-up results on the computational biology benchmark.

Some of the cited storage mapping optimization methods were implemented by their authors, but only applied to small kernels. Indeed, no large-scale implementation of polyhedral techniques for locality improvement have been ever released. This is one of the major goals of a more extensive project in our team: a framework and polyhedral transformation library for iterative and feedback-directed optimization of loop nests [5]. Register blocking is one of the obvious optimizations that such a framework should provide. Implementing a robust array data-flow analysis is the first priority, array expansion, insertion of reuse buffers and storage mapping optimization will follow.

Our work may have applications beyond register promotion. Our contributions should naturally apply to optimizations for random-access memory structures as well, especially those where explicit data transfers are either required (like for register-to-memory transfers) or highly beneficial: scratch-pad or local memories, translation buffers, non uniform shared memories, and disks.

5. REFERENCES


