Towards a Source Level Compiler: Source Level Modulo Scheduling

Yosi Ben-Asher Danny Meisler
Computer Sci. dep.
Haifa University, Haifa.
Email:dmeisler@cs.haifa.ac.il

Abstract

Modulo scheduling is a major optimization of high performance compilers wherein The body of a loop is replaced by an overlapping of instructions from different iterations. Hence the compiler can schedule more instructions in parallel than in the original option. Modulo scheduling, being a scheduling optimization, is a typical backend optimization relying on detailed description of the underlying CPU and its instructions to produce a good schedule. This work considers the problem of applying modulo scheduling at source level as a loop transformation, using only general information of the underlying CPU architecture. By doing so it is possible: a) Create a more retargeble compiler as modulo scheduling is now applied at source level, b) Study possible interactions between modulo scheduling and common loop transformations. c) Obtain a source level optimizer whose output is readable to the programmer, yet its final output can be efficiently compiled by a relatively "simple" compiler.

Experimental results show that source level modulo scheduling can improve performance also when low level modulo scheduling is applied by the final compiler, indicating that high level modulo scheduling and low level modulo scheduling can co-exist to improve performance. An algorithm for source level modulo scheduling modifying the abstract syntax tree of a program is presented. This algorithm has been implemented in an automatic parallelizer (Tiny). Preliminary experiments yield runtime and power improvements also for the ARM CPU for embedded systems.

1 Introduction

This work considers the problem of implementing Modulo Scheduling (MS) [16] at software level rather than implementing it at machine level, as is usually done in modern compilers [12]. The main motivation in doing so is to allow users to view the effect of modulo scheduling at source level, allowing possible interaction with other loop transformations and manual improvements. During experiments, it turned out that in many cases, Source Level Modulo Scheduling (SLMS) improved the execution times even when the underlying compiler used "exact" machine level MS. Consequently, SLMS and machine level MS should co-exist even in a high performance compiler. Thus SLMS is used for two different tasks: optimizing programs at source level along with other loop transformations and as a stand alone optimization complementary to machine level MS.

Basically, MS is one type of solution to the problem of extracting parallelism from loops by "pipelining" the loop's iterations as follows:

$$
\begin{array}{lcl}
 & & S1_0: \quad t = A[0] * B[0]; \\
for(i = 0; i < n; i++) & & for(i = 0; i < n-1; i++) \\
\{ & & \{ \\
S1_i: \quad t = \; A[i] * B[i]; & \longrightarrow & S2_i \quad : \quad s = \; s + t; \\
S2_i: \quad s = \; s + t; & & S1_{i+1}: \quad t = \; A[i+1] * B[i+1]; \\
\} & & \} \\
 & & S2_{n-1}: \quad s = s + t;
\end{array}
$$

Note that after this "pipelining" the dependency between $S1_i$ and $S2_i$ has been eliminated and the new statements $S2_i$ and $S1_{i+1}$ can be executed in parallel (denoted by $S2_i||S1_{i+1}$). [1]

Many techniques have been proposed to approximate the solution to the problem of optimal pipelining of loops iterations by eliminating the maximal number of inter iteration dependencies [3, 23].

A common technique to illustrate MS (very schematically) puts consecutive iterations $i, i+1, \ldots$ shifted by a fixed size (called the Initiation Interval or II [16]) in a 2D table of "rows". The instructions of iteration $i+k$ $(k = 0, 1, 2, \ldots)$ are placed in the k'th column of this table, starting at the $II * k$ row. Let $I_0, \ldots, I_{n-1}$ be the assignments or instructions in the loop's body, then rows $n - II, \ldots, n-1$ will repeat themselves i.e., the instructions in rows $n - II, \ldots, n-1$ will be identical to the instructions in rows $n, \ldots n + II - 1$ and so forth. This repeated II rows form the kernel of the new loop. The first $n - II$ rows form the prologue used to initialize the "iterations pipe" and the last $n - II$ rows (if we put only $2n - II$ iterations) from the epilogue that drains the pipe. The II is valid if the resulting kernel does not violate any data dependency of the original loop. Figure 1 depicts this basic form of MS.

In modern compilers, MS is executed at machine level after the machine depended optimization level. It is natural since in this case the machine instructions of the loop's body fill the columns of the MS table, which forms a schedule of the new loop's instructions. Every row of the table corresponds to instructions that can be executed in parallel:

- For VLIW architectures such as TI, each row of the kernel is a VLS

[1] This parallel execution $S2_i||S1_{i+1}$ is valid under the assumption that in a parallel execution the load of t in $S2_i$ is not affected by the update of t in $S1_{i+1}$. Such a claim is true for most VLIW machines and other models.

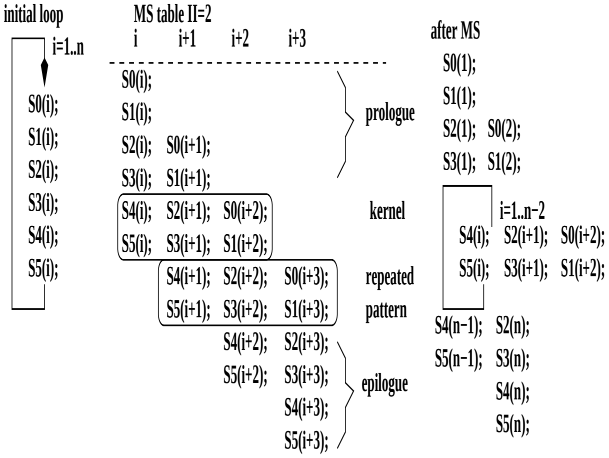


Figure 1: *Using the MS table.*

(compound instruction).

- For super scalar architectures such as the Pentium, each row contains instructions that can be executed in parallel by the different pipeline units of the CPU.

Consequently, each row of the MS table should be valid, in terms of the data dependencies, as well as in order not to violate the amount of hardware resources (and possibly encoding restrictions). For example, if the hardware allows only two parallel additions, any row with more than two additions implies that either the II is wrong or the instructions in the kernel's rows should be rearranged. In addition, MS is also used to minimize the amount of pipeline stalls between the consecutive rows (VLSs in VLIW architectures) of the resulting kernel. Figure 2 depicts MS of a simple loop after it has been compiled to machine code. In this case, the hardware allows VLSs with up to two load/store instructions and up to two additions. The MS table was filled by using $II = 1$ and $(r0), (r0+1), (r0+2), \ldots$ as the iteration index.

This work considers another possibility of implementing MS, namely to

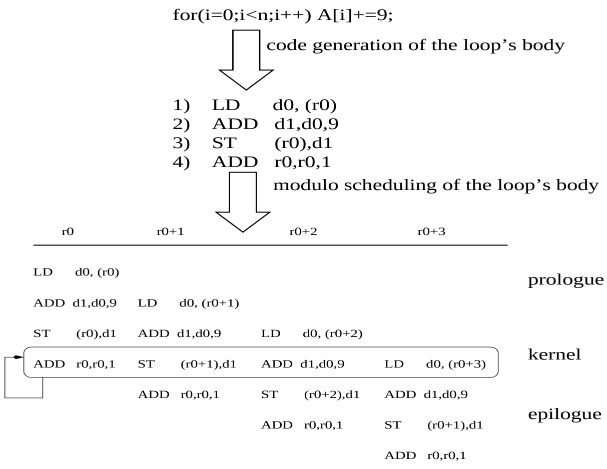
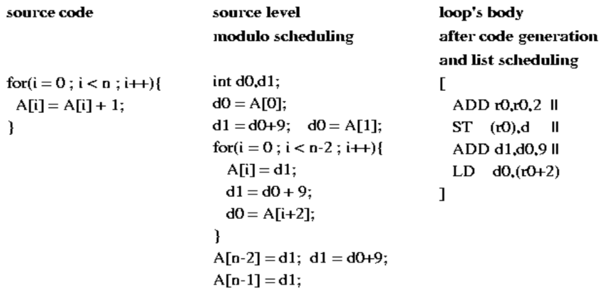


Figure 2: *Machine level MS.*

implement it as a source level loop transformation. The goal is to develop eventually a Source Level Compiler (SLC) that will combine SLMS and known loop transformations such as peeling, fusion, and tiling as described in [4]. A program is first compiled by using the SLC and then the resulting optimized program is compiled to the target architecture by using a regular compiler (called the final compiler).

Figure 3 depicts how SLMS is applied. After SLMS the final compiler applies code-generation, register allocation and list scheduling of basic blocks to create VLIW instructions. The outcome in this case is as efficient as the one would have obtained by using machine level MS. Remark: some MS algorithms such as Iterative MS [17] use modified versions of List scheduling to schedule the kernel after the II has been computed. In this respect, it may be possible to view SLMS as moving the first part of MS to the front-end (computing the II and generating the prologue, kernel and epilogue) leaving the actual scheduling of the kernel to the List scheduling of the backend.



source code

```
for(i = 0 ; i < n ; i++){
  A[i] = A[i] + 1;
}
```

source level
modulo scheduling

```
int d0,d1;
d0 = A[0];
d1 = d0+9;    d0 = A[1];
for(i = 0 ; i < n-2 ; i++){
   A[i] = d1;
   d1 = d0 + 9;
   d0 = A[i+2];
}
A[n-2] = d1;  d1 = d0+9;
A[n-1] = d1;
```

loop's body
after code generation
and list scheduling

```
[
   ADD r0,r0,2  ||
   ST   (r0),d    ||
   ADD d1,d0,9 ||
   LD   d0,(r0+2)
]
```

Figure 3: *Using SLMS followed by List scheduling.*

2 Source level compiler scheme

We show that the SLC can improve final performances of programs (by using advanced array analysis and source level transformations) as follows:

- Based on the interaction with the SLC, the user can modify parts of its code producing new opportunities for the SLC (e.g, replacing while-loops by fixed range for-loops or using arrays instead of pointers/records). The user can acknowledge speculative operations of the SLC such as allowing SLMS to use II that violates some data dependency. The proposed SLMS algorithm is designed to minimize the changes to the original program thus, preserving the readability of the optimized code.

- SLMS is a powerful optimization that can potentially improve the execution times even if the underlying final compiler includes a machine level MS. Thus, the SLC can potentially improve execution times of modern compilers or cover the lack of a given optimization (e.g., MS) in the backend of the final compiler.

- The combination of SLMS and loop transformations can be, in some cases, more effective when it is implemented at source level (as shown later on several possible combinations).

Figure 4 presents a block diagram of the SLC scheme. The programmer interacts with the SLC to improve the performance of his code.

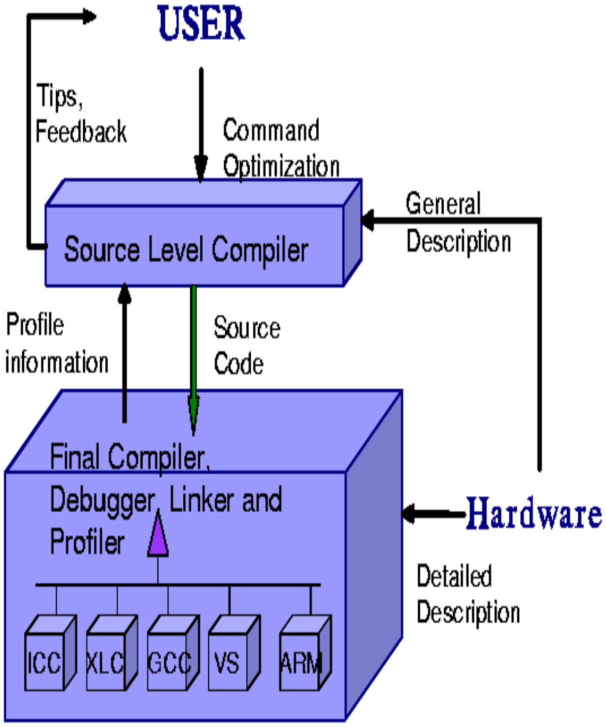


Figure 4: *Source Level Compiler Interaction with the User, Final Compiler and HW.*

Figure 5 is an example of how the SLC can improve the register allocation of the final compiler. This is done in the following steps:

1. The original loop is given as input to the SLC

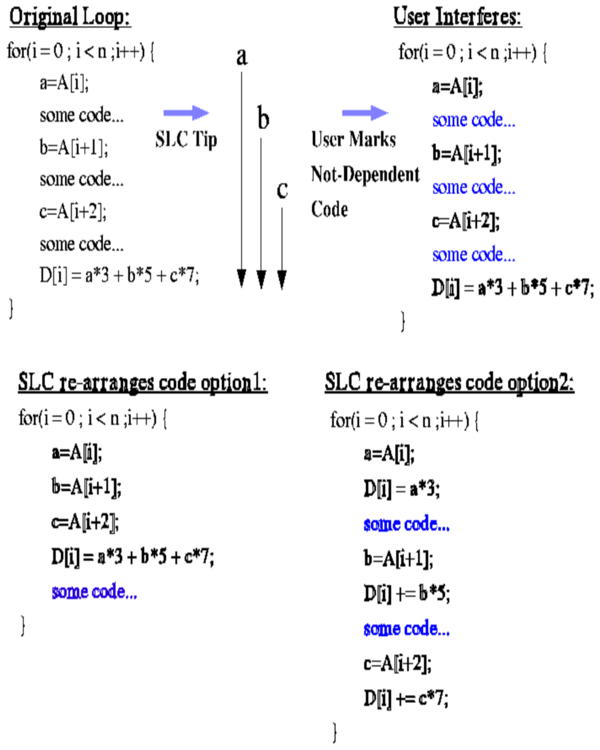
2. The SLC tips the user that the life-times of loop-variants (a,b and c) can be reduced.

3. Than the user marks the code that does not depend on those variables.

4. SLC re-arranges the source code such that the life-times are reduced.

5. The loop is than compiled by the final compiler resulting a better register allocation scheme.

Note that this optimization is usually done by the register allocation of the compiler, apart from cases where the compiler is not able to move instruction due to possible dependencies with the rest of the code. For example of the code after $a = A[i]$ contains a call to a function that may change the value of a than only the user can hint the SLC that $a = a[i]$ can be safely move after this call.



Original Loop:

```
for(i = 0 ; i < n ;i++) {
      a=A[i];
      some code...
      b=A[i+1];
      some code...
      c=A[i+2];
      some code...
      D[i] = a*3 + b*5 + c*7;
}
```

User Interferes:

```
for(i = 0 ; i < n ;i++) {
      a=A[i];
      some code...
      b=A[i+1];
      some code...
      c=A[i+2];
      some code...
      D[i] = a*3 + b*5 + c*7;
}
```

SLC re-arranges code option1:

```
for(i = 0 ; i < n ;i++) {
      a=A[i];
      b=A[i+1];
      c=A[i+2];
      D[i] = a*3 + b*5 + c*7;
      some code...
}
```

SLC re-arranges code option2:

```
for(i = 0 ; i < n ;i++) {
      a=A[i];
      D[i] = a*3;
      some code...
      b=A[i+1];
      D[i] += b*5;
      some code...
      c=A[i+2];
      D[i] += c*7;
}
```

Figure 5: *Source Level Compiler can improve register allocation.*

3 Basic operations used by the SLMS algorithm

In the following subsections, are listed shortly the elementary operations used by the proposed SLMS algorithm. Some of these operations are known and were used in other MS algorithms. Initially, the loops are represented by their abstract syntax tree (AST) [2]. In addition, the dependencies (including the iteration-distances) between array references and scalar variables in the AST are given as directed labeled edges between the AST nodes. For example, the body of the loop $for(i = 0; i < n; i++) A[i] += A[i-1];$ is depicted in figure 6. The input AST is logically partitioned to "multi-instructions"(MI), corresponding to assignments, function-calls or to elementary if-statements. For example the AST in figure 6 contains a single MI.

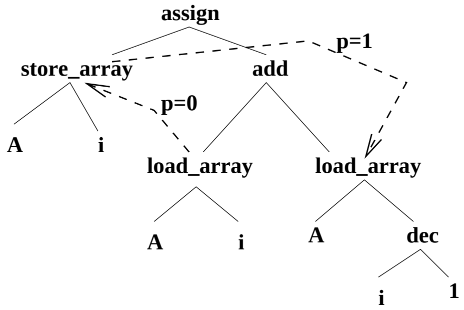


Figure 6: *Input structure for the SLMS algorithm.*

Next, we describe the concept of the minimum initiation interval (MII) [16] and how it is computed. The minimum initiation interval is the one for which a valid schedule exists. Smaller values of II correspond to higher throughput. Calculation of the II accounts for two constrains:

1. Resource constraint (RMII). Let $r(i)$ be the number of available resources (e.g. add units) and $n(i)$ the number of times the resource i is used in the code. $RMII = max_i \lceil \frac{r(i)}{n(i)} \rceil$.

2. Recurrence constraint (PMII) is computed over the data dependency graph G of the loop's body [4]. For a given cycle of dependencies C_i in G let pm_i be the ratio of the sum of delays along C_i and the sum of "iteration-distances" in C_i. The delay (for machine level) between two instructions is basically the number of pipeline stalls that occur if the two instruction are executed one after the other. For SLMS a different notion of delays will be defined as pipeline stalls has no meaning at source level.

   The "iteration-distance" indicates the number of iterations that separate the "define" and "use" of a value (e.g., the iteration-distance between $A[i] = x$ and $y = A[i-3]$) is three).

3. The value of MII is set to $MII = Max\{PMII, RMII\}$.

The MS algorithm first attempts to obtain a valid schedule of with $II = MII$ MIs. In case that such a schedule is not possible the MS algorithm tries larger values of II until such a schedule is obtained.

3.1 Source level if-conversion

MS algorithms are basically designed to work on simple loops without conditional branches. The algorithms that do handle conditional branches usually use predicated instructions to eliminate the conditional branches [19]. In case that the underlying machine does not have predication, the reverse if-conversion is used to restore conditional branches after MS was applied [21]. This work, uses predication at source level. If-statements of the AST are predicated with Boolean variables, similar to the if-conversion operation performed in assembly mode. For example, the if-statement $if(x <$

$y)x = x + 1; A[i] + = x; elsey = y + 1;$ is converted to

$$c = (x \ < \ y);$$
$$if(c) \ x = x + 1;$$
$$if(c) \ A[i] + = x;$$
$$if(not(c)) \ y = y + 1;$$

Remark: apart from the use of if-conversion in MS there have been other proposals for MS of loops with conditional statements. For example, Lam [10] uses a sequence of hierarchical reductions of strongly connected components to MS a loop with conditional statements.

3.2 Decomposition of MIs

This operation divides a complex "large" MI to a set of "smaller" MIs, e.g., $A[i] = a+b*c$; may be divided to $t = b*c$; $A[i] = a+t$;. As explained before, in SLMS the resulting code must be as similar as possible to the original code. Hence, we are seeking to minimize the number of decompositions of MIs needed to obtain a valid SLMS. Finding a minimal decomposition of MIs is a key problem in SLMS and the implemented algorithm uses the following two types of operations:

1. Break a self data dependency edge inside the AST of the MI, e.g. the one between $A[i] + = A[i-1]$;.

2. Reduces the number of resources (arithmetic operations and load/store operations) in the MI. For example the MI $x = A[i]+B[i]+C[i]+D[i]$; contains four load operations and four additions. In assumption that the underlying CPU is a VLIW machine allowing up to two additions and two load/store operation in a multi-instruction (VLS), it is better to decompose $x = A[i] + B[i] + C[i] + D[i]$; to $t = A[i] + B[i]$; and $x = \ t + C[i] + D[i]$.

Decomposition is needed for two reasons:

1. In case that the original loop contains only one MI, at least two are needed to perform MS.

2. In case a loop-carried self dependency prevents finding the MI (section 5).

Consider the loop:

```
$for(i = 0; i < N; i++)\{$
   $A[i] = A[i-1] + A[i-2] + A[i+1] + A[i+2];\}$
```

This loop does not have a valid schedule for $II = 1$, because there is only one MI and because of the loop-carried self dependency between $A[i], A[i-1]$. First, we select one load array reference $A[i+1]$ with no flow dependence with the store operation $A[i] =$. By using this selected array reference we create two MIs using a temporary variable as follows:

```
$for(i = 0; i < (N-2); i++)\{$
   $reg1 = A[i+2];$
   $A[i] = A[i-1] + A[i-2] + A[i+1] + reg1;$
$\}$
```

The data dependency of $reg1 = ...$ and $A[i-2] + reg1+$ will be eliminated by applying Modulo Variable Expansion (MVE), described in section 3.3. At this stage SLMS can be applied with $II = 1$ as follows:

```
$reg1 = A[2];$
$for(i = 0; i < (N-3); i++)\{$
   $A[i] = A[i-1] + A[i-2] + A[i+1] + reg1; ||$
   $reg1 = A[i+3];$
$\}$
$A[i] = A[i-1] + A[i-2] + A[i+1] + reg1;$
```

The symbol || is used between multi instructions that can be totally parallelized by the final compiler/hardware in terms of not violating any data dependencies.

Remark: SLMS assumes that the backend compiler shall use a register for the new local variable "reg1".

3.3 Modulo variable expansion

The SLMS operation, as explained so far can introduce new data dependencies between MIs, such as the dependency between ... $a[i-2]+reg1+a[i+2]$... and ...$reg1 = a[i+2]$; in the last code example of subsection 3.2. Such dependencies may prevent the underlying scheduler (the scheduler of the final compiler) or the hardware (in case of a Super scalar CPU) to extract parallelism. Modulo variable expansion (MVE) [10] is used to eliminate such dependencies. Basically, MVE of a variable (say $reg1$) is performed by unrolling [2] the kernel, and renaming the variable such that the data dependency inside each unrolled copy of the kernel is removed.

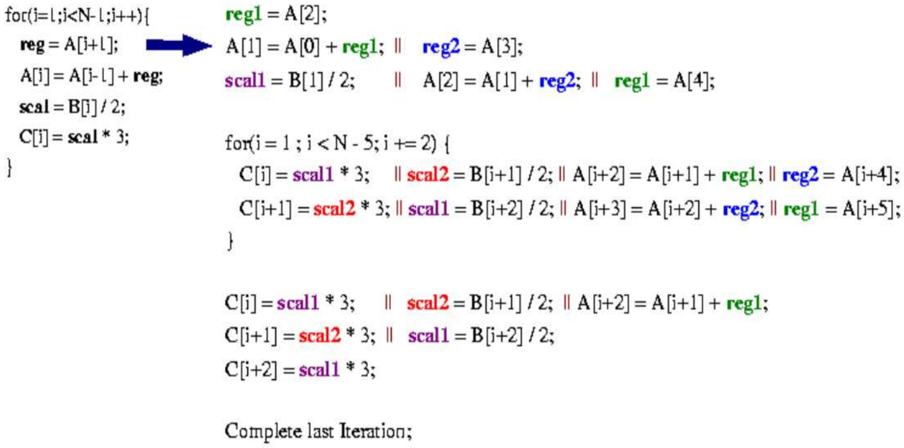
```
$reg1 = a[2];$
$for(i = 0; i < (N-4); i+ = 2)\{$
  $a[i] = a[i-1] + a[i-2] + a[i+1] + reg1; ||$
    $reg2 = a[i+3];$
  $a[i+1] = a[i] + a[i-1] + a[i+2] + reg2; ||$
    $reg1 = a[i+4];$
$\}$
$a[i] = a[i-1] + a[i-2] + a[i+1] + reg1;$
```

Note that after MVE the MIs of each copy (in the unroll operation) can be executed in parallel forming a source level "parallel set of MIs" (indicated

[2] The number of times we need to unroll the loop depends on the lifetime of each variable in the loop as described in [10].

by the || symbol in each row).

The following example (see figure 7) presents an application of SLMS and MVE. In this example the original loop contained a loop variant named scal. The first MI of the loop was decomposed by SLMS generating a second loop variant named reg. MVE was applied separately for each loop variant, generating two registers for each variant.



```
for(i=1;i<N-1;i++){
  reg = A[i+1];
  A[i] = A[i-1] + reg;
  scal = B[i] / 2;
  C[i] = scal * 3;
}
```

```
reg1 = A[2];
A[1] = A[0] + reg1; ||   reg2 = A[3];
scal1 = B[1] / 2;      ||   A[2] = A[1] + reg2; ||   reg1 = A[4];

for(i = 1 ; i < N - 5; i += 2) {
  C[i] = scal1 * 3;    || scal2 = B[i+1] / 2; || A[i+2] = A[i+1] + reg1; || reg2 = A[i+4];
  C[i+1] = scal2 * 3; || scal1 = B[i+2] / 2; || A[i+3] = A[i+2] + reg2; || reg1 = A[i+5];
}

C[i] = scal1 * 3;    ||  scal2 = B[i+1] / 2; || A[i+2] = A[i+1] + reg1;
C[i+1] = scal2 * 3; ||  scal1 = B[i+2] / 2;
C[i+2] = scal1 * 3;

Complete last Iteration;
```

Figure 7: *SLMS decomposition and original loop scalar.*

3.4 Scalar expansion

Another possibility to remove data dependencies caused by scalar variables is to use scalar expansion [4] and replace the scalar variable by a sequence of array references. For example, instead of applying MVE on the loop of section 3.2 scalar expansion can be applied by replacing $reg1$ by $regArr[i]$

so that the SLMS will be:

```
$regArr[2] = a[2];$
$for(i = 0; i < (N - 3); i++)\{$
    $a[i] = a[i-1] + a[i-2] + a[i+1] + regArr[i+2];$
    $||\ regArr[i+3] = a[i+3];$
$\}$
$a[i] = a[i-1] + a[i-2] + a[i+1] + regArr[i+2];$
```

This operation removed the anti-dependence caused by $reg1$ and enables the parallel execution of the two expressions indicated by $||$.

3.5 Delay Calculations

For SLMS the delay between two MIs must be defined in general terms related to the source code rather than the hardware. The delay of a data dependency edge (see figure 6) has been defined so, that the sum of delays along every cycle of dependencies will be greater or equal the number of edges in that cycle. If this condition is not met, some dependency will be violated in the resulting kernel. Let MI_i, MI_j be two MIs connected by a dependency edge $e_{i,j}$ then the $delay(MI_i, MI_j)$ is defined as follows:

1. $delay(MI_i, MI_j) \ = \ 1$ if $i = j$ (loop-carried self dependency).
2. $delay(MI_i, MI_{i+1}) \ = \ 1$.
3. $delay(MI_i, MI_j) \ = \ k$ if $e_{i,j}$ is a forward edge and k is the maximal delay along any path from MI_i to MI_j. Note: j is sequentially ordered after i $i < j$.
4. $delay(MI_i, MI_j) \ = \ 1$ if $e_{i,j}$ is a back edge.

Figure 8 depicts a data dependency graph whose edges are labeled by pairs of $< itr\_distance, delay >$ yielding two cycles: $C1 = c \rightarrow d \rightarrow e \rightarrow f \rightarrow c$

and $C2 = c \rightarrow d \rightarrow f \rightarrow c$. The MII due to $C1$ is $(1+1+1+1)/(2+2) = 1$ while the MII due to $C2$ is $(1+2+1)/2 = 2$. Indeed (as depicted in figure 8), a feasible schedule is obtained for $MII = 2$ and not for $MII = 1$ which violates the backedge from f to c.

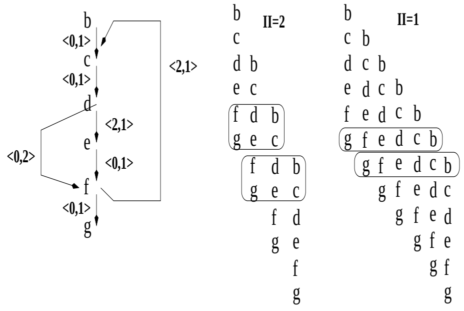


Figure 8: *delays between MIs.*

3.6 Computing the MII

In SLMS the MII accounts only for recurrence constraint (PMII [16]). The computation of the MII is a complex task since the MII is computed over all cycles of dependencies. The Iterative Shortest Path algorithm presented in [3, 23] has been selected for two reasons.

1. First, its simplicity and its ability to naturally handle the case where each dependency edge has several pairs of $< iteration{-}distance, delay >$. This case is frequent in SLMS as each MI may contain more than one array reference, e.g., the edge connecting $MI_i:\ A[i] = B[i-1] + y;$ to $MI_j:\ B[i] = A[i-2] + A[i-3]$ has two iteration distances one for $A[i-2] \longrightarrow A[i]$ and one for $A[i-3] \longrightarrow A[i]$.

2. Second, it does not use the resource MII which is an advantage for SLMS.

4 Filtering Bad-Cases

Filtering "bad cases" where SLMS reduces performance is the first phase of the SLMS algorithm. This phase has to includes various types of heuristics that are specific for both the final compiler and target machine. An example of such a filter is given.

In order to "skip" bad cases, where SLMS reduce performances we compared the ratio between the number of load/store operations (LS) and the arithmetic operations (AO) in the loop's body $\frac{LS}{LS+AO}$. This ratio is termed as the memory-ref ratio. High values of memory-ref implies that overlapping of iterations may lead to too many parallel load store operations in one "row". In that case, SLMS might cause stalls due to memory reference pressure. Experimentaly, it turned out that many such "bad cases" can be eliminated if we require that the above ratio will be less than 0.85. For example, the following loop has $LS = 6$ and $AO = 1$ and ratio 0.857 and thus SLMS will not be applied here.

```
$for(k = 0; k < n; k++)\{$
    $CT = X[k, i];$
    $X[k, i] = X[k, j] * 2;$
    $X[k, j] = CT;$
$\}$
```

Note that if we have several arithmetic operations per each load/store operation then the scheduler can probably hide memory delays (such L1-cache misses) using these arithmetic computations. Remark: Although not tested on other machines, we assume that the memory-ref ratio is machine-specific, and that this ratio depends on the machine's capacity to perform parallel memory operations and the delay of an L1-cache miss. An alternative way of filtering bad cases would have been to estimate the expected number of cycles of the loop's body after SLMS length of the critical path Other factors that can affect this ratio include: penalty of L1-cache misses,

Following is an example showing how SLMS may increase the number of memory references due to overlapping of successive iterations. In the following simplified loop, most array references can be replaced by a register. But, after applying SLMS, the array references must be implemented by separate load/store operations.

$$\begin{array}{lcl}
for(i=0;i<n;i++)\{ & & prologue \\
\quad a[i]+=i; & & for(i=0;i<n-2;i++)\{ \\
\quad a[i]*=6; & \longrightarrow & \quad a[i]--;\ ||a[i+1]*=6;\ ||a[i+2]+=i; \\
\quad a[i]--; & & \} \\
\} & & epilogue
\end{array}$$

5 The SLMS algorithm

The Overall structure of the SLMS algorithm is as follows.

1. A test to filter bad cases where SLMS will probably degrade performances is applied (explained in section 4).
2. Apply software if-conversion.
3. Generate all the MIs in the loop's body, following the order of execution in the source code. Re-name multi defined-used scalars.
4. Find the MII.
   (a) Dependency edges are "raised" to the root of each MI (section 3.6).
   (b) Obtain the delays of the data dependencies edges (section 3.5).
   (c) Compute the MII (section 3.6).

5. If there is no valid MII, then repeat the following until a valid II is obtained or a failure occurs:

    (a) Select [3] a MI and decompose it (section 3.2) based on data dependency analysis. If there are no MIs that can be decomposed then a failure occurs.

    (b) Re-compute delays and MII.

6. If the MII was found, then:

    (a) Update registers lifetime (used for MVE 3.3), save the maximum lifetime.

    (b) Build the prologue kernel and epilogue.

    (c) For each decomposed MI, MVE (section 3.3) or Scalar Expansion (section 3.4) is applied to eliminate dependencies caused by the decomposition. MVE or Scalar Expansion may also be activate to eliminate false dependencies caused by the use of scalars in the loop. The choice between MVE and Scalar Expansion is given to the user as MVE implies loop unrolling and code expansion while Scalar Expansion uses temporary arrays.

Computing MII is performed as follows.

1. initialize the difMin Matrix [3], and obtain delay and flow or anti data dependencies between MIs. Edges connecting memory reference nodes are propagated up to the parent MI.

2. activate the Iterative Shortest Path algorithm [23] with increasing values of II until a valid II is found and returned, or II is equal to the number of MI in the loop, in this case return error.

[3] Selection of a MI can be done by sequential order or by data dependence analysis.

Note, SLMS defines a **valid II** as one that yields a better schedule then the sequential one, e.g. $II < number\ of\ sequential\ MIs.$

Consider the following loop for finding the maximum of an array:

```
$max = arr[0];$
$for(i = 0; i < n; i++)$
  $if(max < arr[i])max = arr[i];$
```

Using source level if-conversion and MVE, the following SLMS was obtained:

```
$max0 = arr[0];$
$max1 = max0;$
$pred0 = (max0 < arr[1]);$
$for(i = 1; i < n - 2; i+ = 2)\{$
  $if(pred0)max0 = arr[i]; ||$
    $pred1 = (max1 < arr[i + 1]);$
  $if(pred1)max1 = arr[i + 1]; ||$
    $pred0 = (max0 < arr[i + 2]);$
$\}$
$if(pred0)\ max0 = arr[i];$
$if(max0 > max1)\ max = max0;\ \ else\ max = max1;$
```

Note: The last line was added manually.

Some loops don't require decomposition of MI nor MVE, such loops have more than one MI and no loop variants. The following example demonstrates such a case. In this loop the lack of loop-carried dependence edges generated a MS with $MII = 1$.

```
$for(ky = 1; ky < n; k++)\{$
    $DU1[ky] = U1[ky+1] - U1[ky-1];$
    $DU2[ky] = U2[ky+1] - U2[ky-1];$
    $DU3[ky] = U3[ky+1] - U3[ky-1];$
    $U1[ky+101] = U1[ky] + 2*DU1[ky] + 2*DU2[ky] + 2*DU3[ky];$
    $U2[ky+101] = U2[ky] + 2*DU1[ky] + 2*DU2[ky] + 2*DU3[ky];$
    $U3[ky+101] = U3[ky] + 2*DU1[ky] + 2*DU2[ky] + 2*DU3[ky];$
$\}$
```

SLMS transformation removed inter-iteration sequential dependencies allowing parallel execution of all MIs within one iteration.

```
$Epilogue...;$
$for(ky = 1; ky < n-5; ky++)\{$
    $U3[ky+101] = U3[ky] + 2*DU1[ky] + 2*DU2[ky] + 2*DU3[ky];\ ||$
    $U2[ky+1+101] = U2[ky+1] + 2*DU1[ky+1] + 2*DU2[ky+1] + 2*DU3[ky+1];\ ||$
    $U1[ky+2+101] = U1[ky+2] + 2*DU1[ky+2] + 2*DU2[ky+2] + 2*DU3[ky+2];\ ||$
    $DU3[ky+3] = U3[ky+3+1] - U3[ky+3-1];\ ||$
    $DU2[ky+4] = U2[ky+4+1] - U2[ky+4-1];\ ||$
    $DU1[ky+5] = U1[ky+5+1] - U1[ky+5-1];\ ||$
$\}$
```

6 SLMS and other loop transformations

SLMS can be combined with other loop reordering and restructuring transformations [4]. At source level, MS can be applied both before or after other loop transformations. The first form of combining is to apply SLMS after loop transformations to extract the parallelism exposed by these transformations. For example, SLMS can not be directly applied to the following inner loop due to the dependency of $a[i,j+1]=t$; and $t=a[i,j+1]$; as depicted by the following erroneous kernel obtained by using $II=1$:

$$\begin{array}{lcl}
for(i=0;i<n;i++) & & \\
\quad for(j=0;j<n;j++)\{ & & t=a[i][j]; \\
\quad\quad t=a[i][j]; & \longrightarrow & a[i][j+1]=t;\ ||\ t=a[i][j+1]; \\
\quad\quad a[i][j+1]=t; & & \qquad\qquad\qquad a[i][j+2]=t; \\
\quad \} & &
\end{array}$$

Using loop interchange [4] to replace the innermost loop from $'j'$ to $'i'$ yields a legal kernel with $II=1$. Note that the dependence on the temporary variable t is resolved by using MVE. This allows the parallel execution of MI separated by $||$.

$$\begin{array}{lcl}
 & & for(j=0;j<n;j++)\{ \\
 & & \quad t1=a[0,j]; \\
for(j=0;j<n;j++)\{ & & \quad for(i=0;i<n-2;i+=2)\{ \\
\quad for(i=0;i<n;i++)\{ & & \quad\quad a[i,j+1]=t1;\ ||\ t2=a[i+1,j]; \\
\quad\quad t\ =\ a[i,j]; & \longrightarrow & \quad\quad a[i+1,j+1]=t2;\ ||\ t1=a[i+2,j]; \\
\quad\quad a[i,j+1]=t; & & \quad \} \\
\quad \} & & \quad a[i,j+1]=t1; \\
 & & \}
\end{array}$$

Performing MS at source level enables its application also before other loop transformations. Another example where loop transformations allow us to apply SLMS is loop fusion [4]. Each of the following two loops can not be SLMSed due to the dependency between the first statement of the next iterations and the last statement of the current iteration. After loop fusion we get a single loop, now SLMS can be applied obtaining a valid scheduling with $II = 3$ as follows:

```
$for(i = 1; i < n; i++)\{$
    $t = A[i-1];$
    $B[i] = B[i] + t;$
    $A[i] = t + B[i];$
    $\}$
$//second\ loop$
$for(i = 1; i < n; i++)\{$
    $q = C[i-1];$
    $B[i] = B[i] + q;$
    $C[i] = q * B[i];$
$\}$
$TR$
```

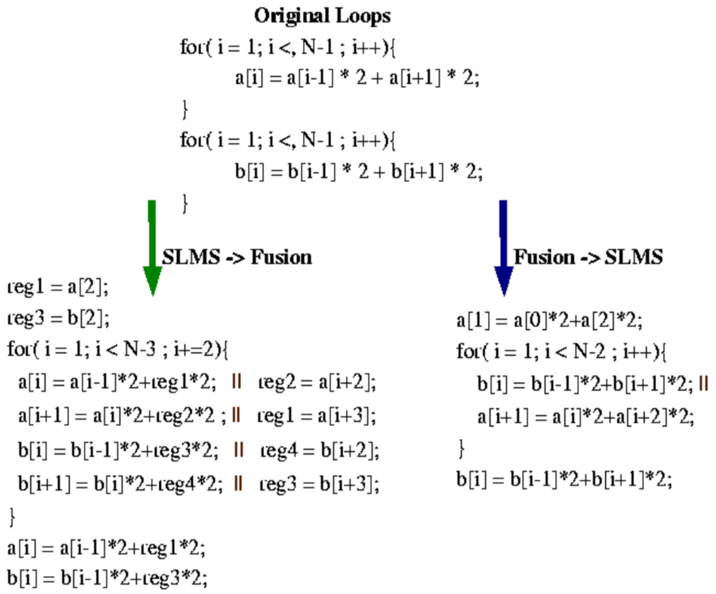
```
$for(i = 1; i < n; i++)\{$
    $t = A[i-1];$
    $B[i] = B[i] + t;$
    $A[i] = t + B[i];$
    $q = C[i-1];$
    $B[i] = B[i] + q;$
    $C[i] = q * B[i];$
$\}$
```

```
$t = A[i-1];$
$B[i] = B[i] + t;$
$A[i] = t + B[i];$
$q = C[i-1];$     $||t = A[i];$
$B[i] = B[i] + q;$ $||B[i+1] = B[i+1] + t;$
$C[i] = q * B[i];$ $||A[i+1] = t + B[i+1];$
                   $q = C[i];$
                   $B[i+1] = B[i+1] + q;$
                   $C[i+1] = q * B[i+1];$
```

Consider two loops, applying SLMS separately to each loop followed by Fusion of the two loops will generate a different schedule than first applying Fusion and then SLMS to the fused loop. The example depicted in figure 9 demonstrates this case.

SLMS can also be used to enable the application of loop transformations. For example, the following two loops can not be joined by loop fusion. Usually, this example is solved using a complex combination of loop peeling + loop reversal, however one application of SLMS (as depicted in figure 10) will allow loop fusion.

Loop unrolling is used to resolve cases where the II is to high (close to the number of MI). Also, in some cases, unrolling the kernel of an SLMSed loop can improve resource utilization. In conclusion, clearly there are cases where the combination of loop transformations and SLMS is useful.



```
for( i = 1; i <, N-1 ; i++){
        a[i] = a[i-1] * 2 + a[i+1] * 2;
}
for( i = 1; i <, N-1 ; i++){
        b[i] = b[i-1] * 2 + b[i+1] * 2;
}
```

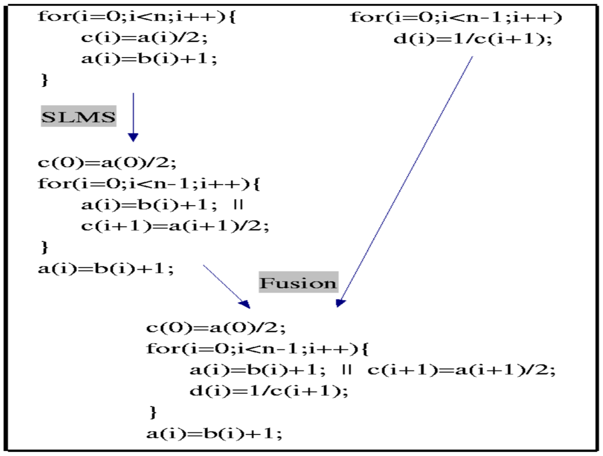
```
reg1 = a[2];
reg3 = b[2];
for( i = 1; i < N-3 ; i+=2){
  a[i] = a[i-1]*2+reg1*2;  ||  reg2 = a[i+2];
  a[i+1] = a[i]*2+reg2*2 ; ||  reg1 = a[i+3];
  b[i] = b[i-1]*2+reg3*2;  ||  reg4 = b[i+2];
  b[i+1] = b[i]*2+reg4*2;  ||  reg3 = b[i+3];
}
a[i] = a[i-1]*2+reg1*2;
b[i] = b[i-1]*2+reg3*2;
```

```
a[1] = a[0]*2+a[2]*2;
for( i = 1; i < N-2 ; i++){
   b[i] = b[i-1]*2+b[i+1]*2; ||
   a[i+1] = a[i]*2+a[i+2]*2;
}
b[i] = b[i-1]*2+b[i+1]*2;
```

Figure 9: *The order of transformations changes the final scheduling.*

7 Cases where SLMS optimizes better than the lower level MS

In here we consider possible explanations of why SLMS can in some cases obtain better schedulings than the underlying lower level MS. First it is important to understand the difference between optimizing at source level mode and at machine level mode. At machine level the optimization can use exact knowledge of the CPU resources and obtained optimized scheduling. The opposite is true for source level optimization which is actually performed ignoring hardware resources constrains, optimizing for maximal parallelism at source level. This "disadvantage" can work to the benefit of a SLMS. In particular, it can happen that due to hardware resource constrains the underlying MS will not optimize a given loop while after SLMS a more optimized scheduling will be obtained. Typically, even an elementary list



```
for(i=0;i<n;i++){
    c(i)=a(i)/2;
    a(i)=b(i)+1;
}
```

```
for(i=0;i<n-1;i++)
    d(i)=1/c(i+1);
```

```
c(0)=a(0)/2;
for(i=0;i<n-1;i++){
    a(i)=b(i)+1; ||
    c(i+1)=a(i+1)/2;
}
a(i)=b(i)+1;
```

```
c(0)=a(0)/2;
for(i=0;i<n-1;i++){
    a(i)=b(i)+1; || c(i+1)=a(i+1)/2;
    d(i)=1/c(i+1);
}
a(i)=b(i)+1;
```

Figure 10: *SLMS allows loop fusion.*

scheduling of basic blocks applied after SLMS can in some cases find better scheduling than the more constrained machine level MS.

We mainly consider the Iterative MS as presented in [18] (IMS) however the following is also valid for other types of MS. The IMS has a complete knowledge of the available hardware resources and once the II has been computed it tries to schedule the kernel's instructions in a modulo reservation table (RT) with II rows. The filling of the RT rows is done following the original instruction order mixing instructions from consecutive iterations $i, i+1, i+2, ...$ The instruction are placed in the RT "as is" relaying on the epilogue to create the necessary pipeline chain. Figure 11 demonstrates a case where the IMS may fail due to increased register pressure. The DDG in figure 11 contains three instructions x, y, z and is frequently found in loop accessing arrays. For example the long delay between x and y can be the result of a more complex arithmetic operation such as floating multiplication, while the dependency cycle between y and z can be easily generated by the index increment of array accesses $(y = ...z[i-1])$ or by an accumulator instruction $(y+ = z[i])$. In this case $II = 2$ and we assume the the IMS is able to build the corresponding kernel (figure 11 left). The use of such a kernel implies that the last four values of x must be held in four different registers since they must remain alive to be used by later iterations of this kernel. As explained before, modulo variable expansion will unroll the kernel four times in order to let the value computed by the x instruction stay alive during the next four iterations. This unrolling increases the register pressure and may lead to performance degradation or the compiler will prevent from using the code generated by the IMS+Modulo-variable-expansion. On the other hand SLMS can be applied to this loop leading to the kernel $[z||x];[y]$ which can be safely schedule (figure 11). Basically the SLMS in this example was used only to expose the possible parallelism of $[z||x]$.

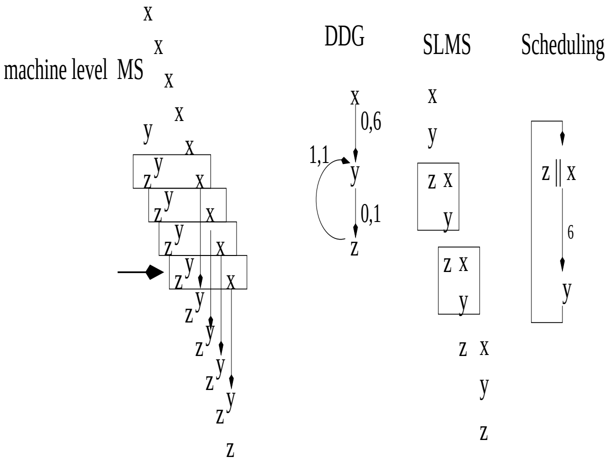


Figure 11: *Failure due to register pressure.*

Another drawback of the IMS is that it can not explicitly correct the indexes of instructions that are placed in the RT. Note that when the k'th instruction is scheduled in a RT with $II < k$ rows it is assumed to belong to the $i + \lfloor k/II \rfloor$ iteration where i marks the iteration of the first instruction. Hence the IMS can not violate the order of the instruction scheduling. The code in the example in figure 12 is taken from [18] where it is used to show how IMS fails to schedule $A3$ and $A4$ in a RT with $II = 4$ rows (figure 12 left). Since SLMS ignores the issue of hardware resources is will produce the kernel $[A3_i||A1_{i+1}];[A4_i||A2_{i+1}]$ which can be schedule (using list scheduling) in a RT with 4 rows (as depicted in figure figure 12 right). Technically the IMS failed since $A3$ and $A4$ must be scheduled to the rows already occupied by $A1$ and $A2$. Note that even if the IMS would have considered mixed solutions such as the one described in figure 12 (right) it lacks the ability of changing the indexing of $A1$ $A2$ from i to $i+1$. Thus this failure of the IMS is not a technical issue it follows from the fact that unlike IMS, SLMS can change the index of instructions while scheduling them in the II rows of the kernel, e.g., from $A1 : r1 = r0 + x[i]$ to $A1 : r1 = r0 + A[i+1]$.

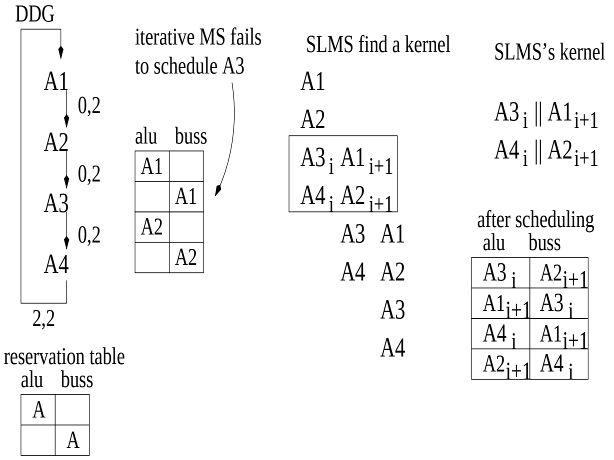


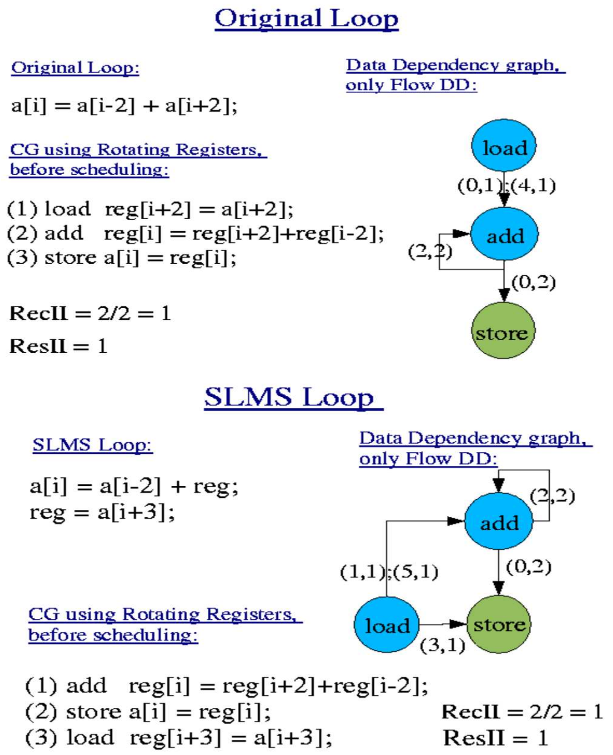
Figure 12: *Failure due to in ability to change indexing of instruction during scheduling.*

Finally, SLMS usually changes the data dependencies of the loop's body compare to the original code thus allowing different (possibly better) schedulings not available in the original code. As an example consider the loop $a[i] = a[i-2] + a[i+2]$; of figure 13 where the code generation used rotating registers [9] to create the loop's code. The underlying MS parallelizes the loop ($MII = RecII = 1$) due to, the dependency cycle between the "load" and the "add". Note, that the "add" was assigned a delay of 2 cycles. The Data Dependency (DD) edges between the "load" and the "add" and not between the "load" and the "store" are due to the use of rotating registers. In addition, redundant "load" optimization was applied (no need to "load" $a[i-2]$). Next, SLMS was applied before code generation obtaining the loop $a[i] = a[i-2] + reg; \quad reg = a[i+3];$. Due to simplicity MVE was not applied. After SLMS, the DD graph for the SLMSed loop (we present only "flow" DD arcs) changes. The MII calculated by the underlying MS remains 1. But since the DD graph changed, the underlying scheduler can generate a different schedule for that loop. Since the scheduler has now more options,the new schedule can be better than the original one. However, note that, any form of parallelization obtained by a machine level MS is clearly obtainable using SLMS, as SLMS is less restricted than machine level MS (limited from resource constrains).

Apart from this ability of SLMS to find optimized scheduling by first ignoring resource constrains, there are some technical factors working in favor of SLMS.

1. It is common that compilers restrict MS to small size loops such as loops with less than 50 instructions. Thus SLMS can optimize and parallelize even large size loops improving their final scheduling however there is no special benefit in applying loop unrolling before SLMS.

2. SLMS works at source level thus can directly determine the exact dependencies between each two array references. Though a compiler can also obtain these dependencies at the front-end/AST level it may fail to transfer them to the machine level representation (RTL) of the back-end. Thus, MS operations such as replacing $A[2*i]$ by $A[2*(i+1)]$



Original Loop

Original Loop:

```
a[i] = a[i-2] + a[i+2];
```

CG using Rotating Registers, before scheduling:

```
(1) load  reg[i+2] = a[i+2];
(2) add   reg[i] = reg[i+2]+reg[i-2];
(3) store a[i] = reg[i];
```

RecII = 2/2 = 1

ResII = 1

Data Dependency graph, only Flow DD:

SLMS Loop

SLMS Loop:

```
a[i] = a[i-2] + reg;
reg = a[i+3];
```

CG using Rotating Registers, before scheduling:

```
(1) add   reg[i] = reg[i+2]+reg[i-2];
(2) store a[i] = reg[i];
(3) load  reg[i+3] = a[i+3];
```

RecII = 2/2 = 1

ResII = 1

Data Dependency graph, only Flow DD:

Figure 13: *SLMS changes the DD graph thus enabling other scheduling options.*

are more complicated to implement in RTL/machine level than at source level.

8 Working with the source level compiler

In here we shortly demonstrate how the user can use the source level compiler (SLC) to on-line improve its source code such that SLMS can be applied. Consider the following loop for which the SLMS obtained a MS with $II = 2$. Based on the outcome, the user can determine that $II = 1$ was not obtained due to a dependency cycle with $temp- = x[lw] * y[j]$ of the next iteration and $lw++$ of the current iteration.

$$\begin{array}{l} lw = 6; \\ for(j = 4; j < n; j = j + 2) \\ \{ \\ \quad temp- = x[lw] * y[j]; \\ \quad lw + +; \\ \} \end{array} \quad \longrightarrow \quad \begin{array}{l} lw = 6; \\ reg1 = y[4]; \\ temp = temp - x[lw] * reg1; \\ for(j = 4; j < n - 1; j = j + 2) \\ \{ \\ \quad lw + +; \ || \ reg1 = y[j + 1]; \\ \quad temp = temp - x[lw] * reg1; \\ \} \\ lw + +; \end{array}$$

The user can fix this problem by moving the $lw++$ before the first MI allowing the MVE to operate replacing lw by two variables $lw1, lw2$. The outcome is that SLMS now obtains a schedule with $II = 1$ increasing the parallelism:

$$\begin{array}{l} lw = 5; \\ for(j = 4; j < n; j = j + 2) \\ \{ \\ \quad lw + +; \\ \quad temp- = x[lw] * y[j]; \\ \} \end{array} \quad \longrightarrow \quad \begin{array}{l} lw1 = 4; \\ lw2 = 5; \\ lw1 + +; \\ for(j = 4; j < n - 2; j = j + 4) \\ \{ \\ \quad temp- = x[lw1] * y[j]; \ || \ lw2 + +; \\ \quad temp- = x[lw2] * y[j + 2]; \ || \ lw1 + +; \\ \} \\ temp- = x[lw1] * y[j]; \end{array}$$

An even better improvement would have been obtained had the user decided to apply manual decomposition of $temp- = x[lw] * y[j]$ before moving $lw++$. Since the lifetime of lw after SLMS is two iterations, then MVE will unroll twice and use renaming to obtain the following code.

$$\begin{array}{l} lw = 5; \\ for(j = 4; j < n; j = j + 2) \\ \{ \\ \quad lw++; \\ \quad reg1 = y[j]; \\ \quad temp- = x[lw] * reg1; \\ \} \end{array} \longrightarrow \begin{array}{l} lw1 = 4;\ lw2 = 6;\ lw3 = 4; \\ lw1++;\ reg1 = y[4]; \\ for(j = 4; j < n - 6; j = j + 6) \\ \{ \\ \quad temp- = x[lw1] * reg1;\ \ ||\ reg2 = y[j + 2];\ \ ||\ lw3+ = 3 \\ \quad temp- = x[lw2] * reg2;\ \ ||\ reg1 = y[j + 4];\ \ ||\ lw1+ = 3 \\ \quad temp- = x[lw3] * reg1;\ \ ||\ reg2 = y[j + 6];\ \ ||\ lw2+ = 3 \\ \} \\ temp- = x[lw1] * reg1;\ \ ||\ reg2 = y[j + 2]; \\ temp- = x[lw2] * reg2; \end{array}$$

9 Experimental results

SLMS was implemented in Wolfe's Tiny system [22] enhanced by the Omega test [14]. Tiny, was chosen, due to its support in source-to-source transformations and its support of array analysis. Tiny is a loop restructuring and research tool which interacts with the user. Tiny's GUI allows the user to select which transformation to apply, it includes among others, Distribution, Interchange, Fusion, Unroll and SLMS. The following benchmarks were used to test SLMS: The NAS [5] benchmark, Livermore [11] loops, Linpack [6] loops, and the STONE benchmark. The benchmarks were compiled and tested using several commercial compilers and machines: Intel's ICC-ia64(V 9.1) and GCC-ia64 over Itanium II (IA64), IBM's XLC over Power 4 Regata, and GCC over ARM simulator. We have also tested SLMS with GCC over superscalar processor Pentium(R). The Experimental results are divided into three subsections: the first describes the results with GCC and the second describes the results obtained using ICC and XLC, and the third describes results for embedded systems. The GCC has a weak Swing MS and thus modeling the use of a general source level compiler optimizing the program (with SLMS) before it is compiled by the relatively weak compiler. ICC and XLC are high performance compilers with advanced machine level MS, their results support the claim that SLMS is a separate optimization that can be used before low level MS is applied. Remarks: (1) in all the following graphs, the Y axes represents the speedup obtained by SLMSed loop vs. non SLMSed loops. In all tests both SLMSed and non SLMSed loops are compiled with the same compilation flags. (2) SLMS was tested with and without source level MVE, the presented results show the best time obtained. (3) In ia-64 architecture, improvement can be measured by counting the number of bundles in the loop body, a bundle can be viewed as a VLS regarding for explicit instruction level parallelism.

9.1 Experimental results over a relatively weak compiler

As explained in the introduction, SLMS is considered as part of a potential SLC. Thus, showing that SLMS improves execution times over GCC supports the claim that a SLC can be used to improve execution times over relatively weak final compilers. The following graphs 14 and 15, present speedups obtained using GCC (IA64) over Itanium II with and without $-O3$. Analyzing GCC's assembly for $-O3$ revealed that scheduling optimizations such as MVE and Unrolling where not performed. In some successful cases such as ddot2 the application of those transformations at source level compensated for the lack of them in the final compiler. Another successful loop is kernel 8, this loop has a big loop body without loop-carried dependency edges and contains only array references. For this kind of loop, SLMS doesn't need to decompose and in this case $MII = 1$. The application of SLMS released the intra-iteration sequential dependency between MI and revealed the parallelism between them, thus enabling the generation of less bundles. Indeed before SLMS GCC's assembly contained 23 bundles and after SLMS 16 bundles.

Regarding bad cases, most of them are within the Linpack loops. Most of those loops contain one long MI and use intensive floating point calculations. The negative results can be explained by the level of parallelism of floating point operations in the Itanium processor. To prove this, we replaced all the floating point variables with integer ones and re-run the test. The results where reversed in favor of SLMS. Another prove is by the fact that those same loops have better speedups on Pentium(R) and Power4-Regata. Filtering bad cases is an important issue in SLMS. Bad cases can be identified at source level by general high level characteristics, experimental results prove that they are specific for the pair compiler/hardware.

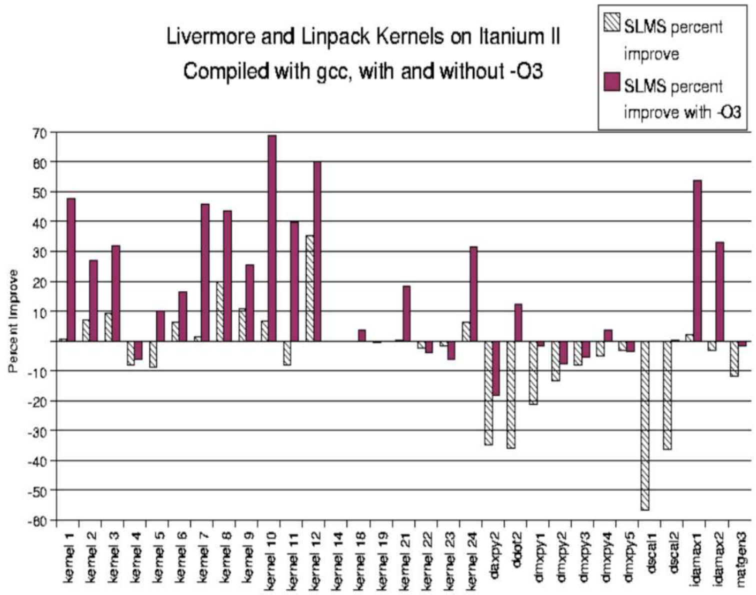


Figure 14: *Livermore & Linpack over GCC*

Another interesting experiment is to see how SLMS as a SLC can be used to close the gap between using and not using -O3 for example in the

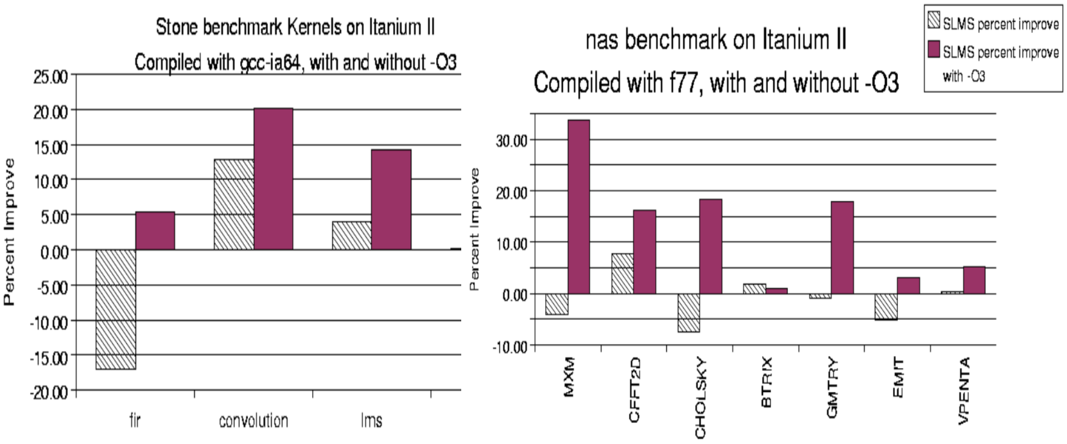


Figure 15: *Stone and NAS over GCC*

ICC compiler. If SLMS can cover a significant part of this gap, it can cover up cases where the underlying final compiler fails to optimize for new architectures. Thus increasing the retargibility of the underlying final compiler. In order to see this, we have compared how SLMS without -O3 can bridge the gap between using -O3 and the relative weak compiler obtained when -O3 is not used. Figure 16 depicts the results over ICC+Itanium, showing that using SLMS without -O3 as a SLC can “close” the gap between a good highly optimizing compiler and a relative weak compiler.

We also tested SLMS on a superscalar processor (Pentium(R)) where all the parallelism is obtained by the HW pipeline. Figure 17 depicts the results, the loops where compiled using GCC with and without $-O3$. The results show that SLMS was successful in exposing the parallelism in most of the loops. One example for which SLMS had a negative impact is kernel 10. Kernel 10 contains several loop-variants and a big loop body causing SLMS's MVE to use 35 register, apparently causing spilling since Pentium(R) has much less registers.

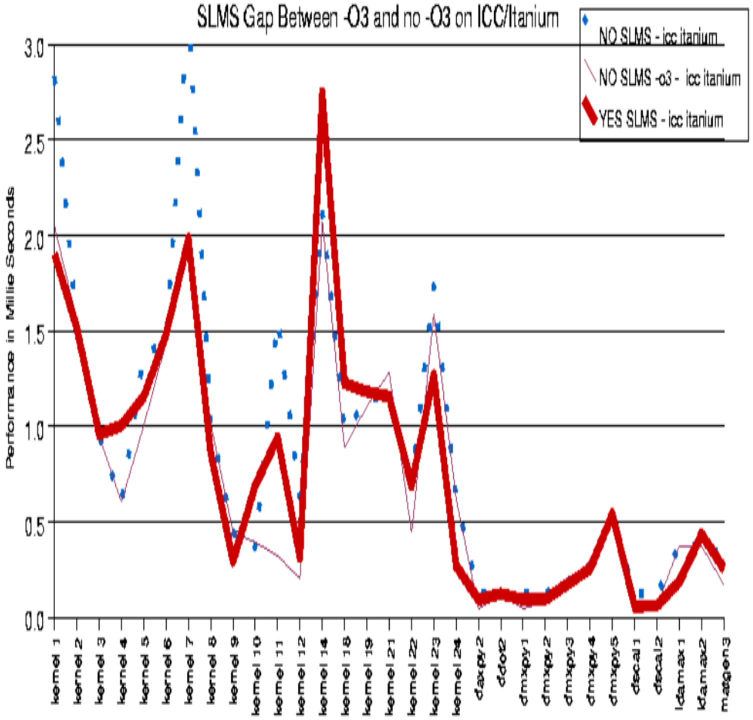


Figure 16: *SLMS can be used to close the gap between using and not using -O3.*

9.2 Experimental results over highly optimizing compilers

The following graphs 18, 19 and 20, present speedups obtained using ICC (IA64) over Itanium II and XLC over Power 4 Regata. Showing that SLMS improves performance over highly optimizing compilers and powerful machines, proving that SLMS should co-exist with low level MS. Another indication to the fact that SLMS can co-exist with low level MS is that out of 31 loops that were tested, ICC performed MS both before and after SLMS for 26 of those loops. For three loops (kernels 2,7 and 24), ICC did not apply MS but SLMS did resulting in positive speedups. For two loops (idamax2

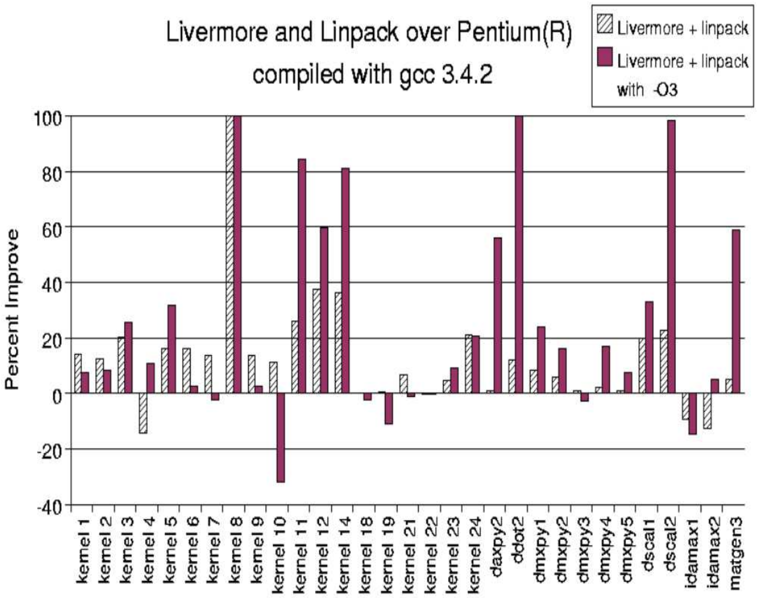


Figure 17: *SLMS can improve performance over superscalar processor.*

and kernel 8), ICC performed MS only before SLMS. SLMS prevented MS of those loops, kernel 8 achieved speedup of almost 15 percent while idamax2 had a negative of the same amount. Showing that SLMS should be selectively applied.

In the following example we analyze a loop that has an intensive floating point computation.

$$float\ k[n];$$
$$for(k = 1; k < n; k++)$$
$$\{$$
$$X[k] = X[k-1] * X[k-1] * X[k-1] * X[k-1] * X[k-1] +$$
$$X[k+1] * X[k+1] * X[k+1] * X[k+1] * X[k+1];$$
$$\}$$

The loop was transformed using SLMS and MVE and compiled with ICC

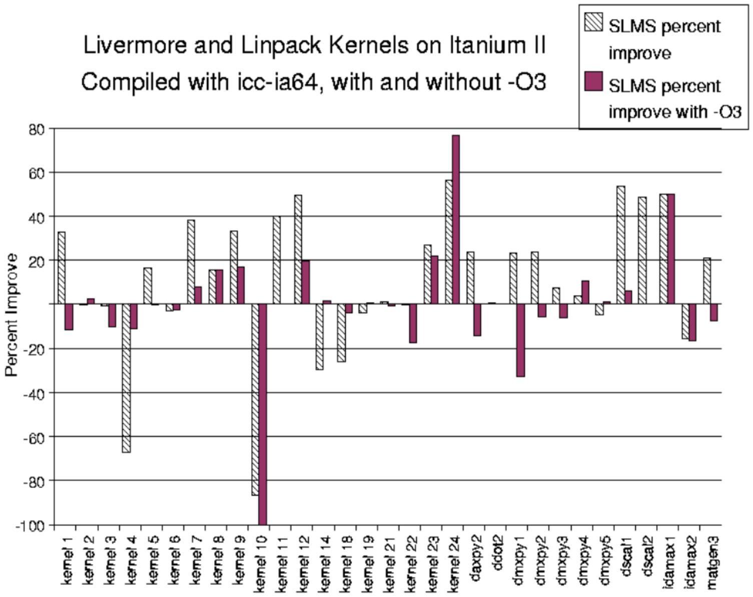


Figure 18: *Livermore & Linpack over ICC*

$-O3$ over ItaniumII.

```
$float\ k[n];$
$reg1 = X[1];$
$for(k = 1; k < n - 3; k+ = 2)$
${$
   $X[k] = X[k-1] * X[k-1] * X[k-1] * X[k-1] * X[k-1]+$
          $reg1 * reg1 * reg1 * reg1 * reg1;\quad || \quad reg2 = X[k+2];$
   $X[k+1] = X[k+1] * X[k+1] * X[k+1] * X[k+1] * X[k+1]+$
          $reg2 * reg2 * reg2 * reg2 * reg2;\quad || \quad reg1 = X[k+3];$
$}$
$X[k] = X[k-1] * X[k-1] * X[k-1] * X[k-1] * X[k-1]+$
$reg1 * reg1 * reg1 * reg1 * reg1;$
```

Since the ItaniumII has two floating point units, and can concurrently execute two bundles, each bundle can contain one fma.s (floating multiply

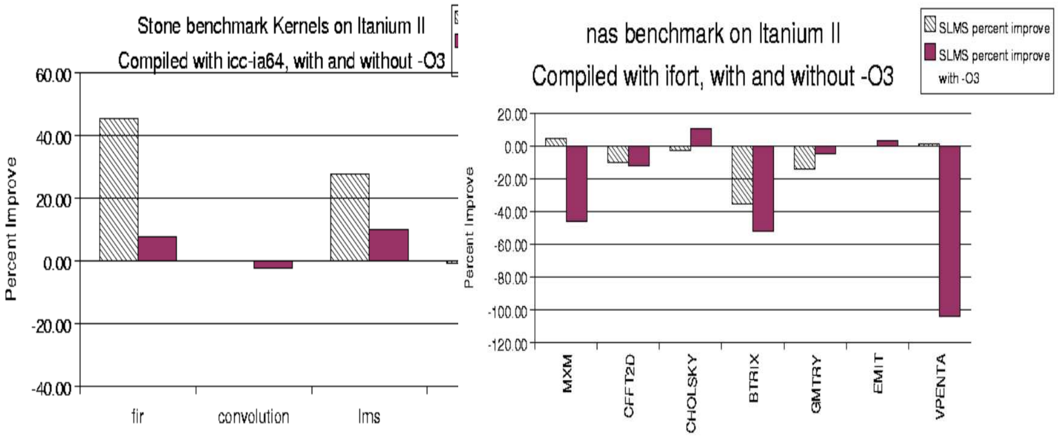


Figure 19: *Stone and NAS over ICC*

and add) instruction. For the original loop ICC unrolled the kernel 8 times until maximum resource utilization was achieved, ICC achieved 5.8 bundles per iteration. SLMS aided ICC to produced a compact and optimized code with 4 bundles per iteration. For both loops ICC performed MS, but the II for the SLMS loop was much smaller than the one for the original loop. This example shows that SLMS can aid the low level MS to find a better solution. This specific example is also relevant for improvement of floating point numeric applications. Livermore kernel 24 contains a condition branch. For both loops (original and SLMSed) ICC did not unroll nor performed MS. For the original loop ICC generated 5 bundles per iteration and for the SLMS loop it generated 3.5 bundles per iteration. This improvement was because SLMS transformed the loop in a way that gave ICC other scheduling options. Apparently, unrolling and mixing iterations enabled ICC to better utilize resources.

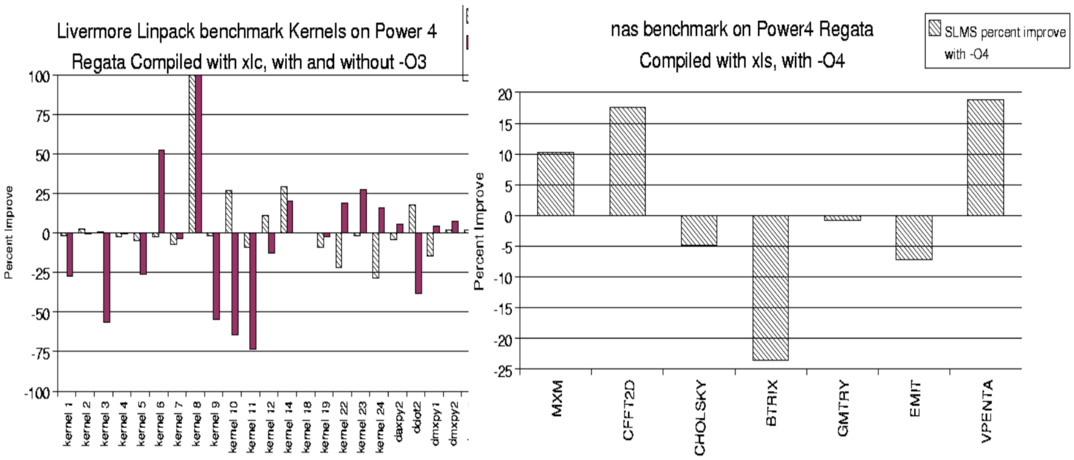


Figure 20: *Livermore & Linpack + NAS over XLC*

9.3 Experimental results for embedded systems

In order to test the effectiveness of SLMS for embedded systems, one should test the power consumption gain/loss involved with SLMS. Moreover, the comparison should be made over a classic embedded core such as the ARM or over a VLIW machine. [4] The effectiveness of SLMS for VLIW machines has been demonstrated by the experiments over the IA-64. The Panalyzer system [1] with the simple-scalar tool chain for ARM is used to measure the effect of SLMS on the power dissipation of the ARM 7TDMI processor. Figure 21 depict the improvements obtained in the overall power dissipation including caches and memories. The results show that SLMS can indeed improve the power dissipation, but not in all cases, hence SLMS must be applied selectively. Similar, results where also obtained for cycle count figure 22. There is a clear correlation between the bad cases of the power consumption and the cycle count. in addition the results over the ARM are worse than those obtained over other architectures. The main reason is

[4] SLMS has a very minor effect on the code size, and thus this aspect of embedded systems has not been considered.

that the ARM does not use Instruction Level Parallelism using basically one ALU operation per cycle. Consequently, the parallelism that SLMS created could only be used for hiding memory latencies and pipeline stalls (compare to the IA64 where it was used to fill empty slots). Thus, the results of figure 21 should be regarded as a success, provided that SLMS will be used selectively.

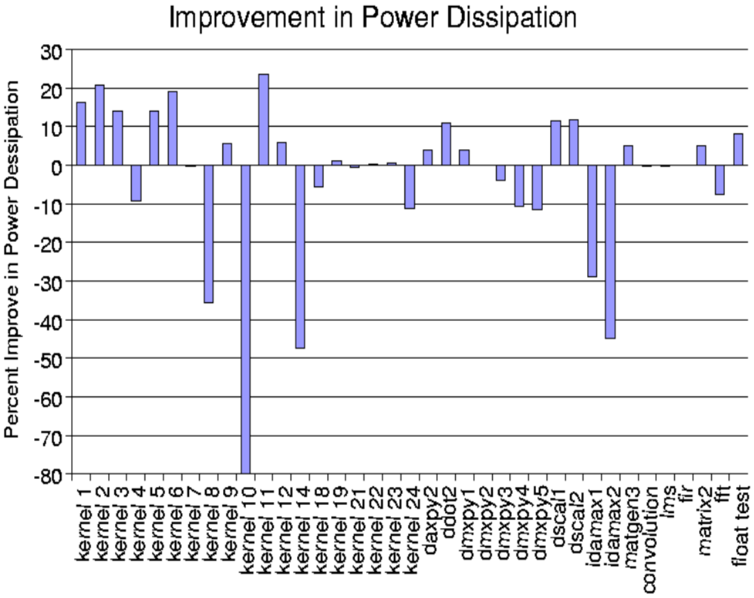


Figure 21: *Power dissipation for the ARM*

10 Possible extensions

In here possible extensions to SLMS are considered, showing its generality of handling more complex cases than the simple loops presented so far. These extensions include applying SLMS to while-loops and applying SLMS to loops with conditional statements. The potential of SLMS to handle while-loops and conditional statements is only demonstrated via examples, full

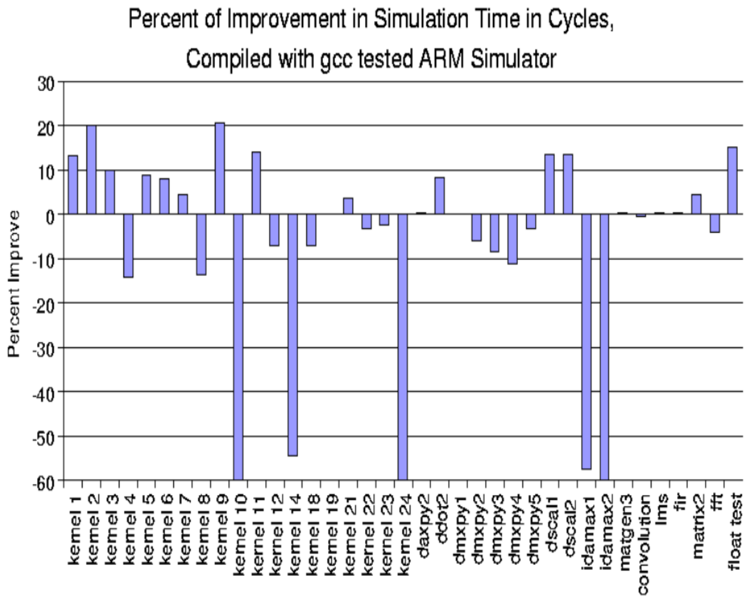


Figure 22: *Total number of cycles for the ARM*

implementation of these extensions is beyond the scope of this work.

It is well known [8] that in some cases while-loops can be unrolled in spite of the fact that their iteration count is not fixed. The ability to unroll while-loops suggests that SLMS can also be applied to while-loops. Following are two examples showing how SLMS can be applied to while-loops.

In the first example, the loop finds the first element in a linked list, whose value equal a given key.

$for(p = head; p = null; p = p- > next)\{$

$\quad if(p- > key == KEY)\ break;$

$\}$

This loop can be unrolled as follows.

```
for(p = head; p= null && p−>next= null; p = p−>next−>next){
  if(p−>key == KEY)break;
  if(p−>next−>key == KEY)break;
}
p =  (p−>key == KEY)?p : p−>next;
```

A kernel can be obtained by overlapping successive iterations as follows.

$iteration\ i$	$iteration\ i+1$	$iteration\ i+2$
$c = (p-\!> key == Key)$		
	$if(c)break;$	$(c = p-\!> next-\!> next) == Key$
		$if(c)break;$

The final SLMS version of this loop is as follows.

```
(c1 = head)= nuu?head−>key == KEY  :  false;
for(p = head; p= null; p = p−>next−>next){
  [if(c1)break; || c2 = (p−>nextSLMS= null)? p−>next−>key == KEY  :  true;]
  [if(c2)break; || c1 = (p−>next−>next= null)? p−>next−>next−>key == KEY  :
}
if(c2) result  =  p−>next;
else if(c1) result  =  (head−>key == KEY)?head  :  p;
else result  =  null;
```

In the second example, the loop performs a shifted copy of a string.

```
i = 0;
while(a[i + 2]){
  a[i] = a[i + 2];
  i + +;
}
```

This loop can be unrolled as follows.

```
$i = 0;$
$startUpCode();$
$while(a[i + 2]$ && $a[i + 3])\{$
   $a[i] = a[i + 2];$
   $a[i + 1] = a[i + 3];$
   $i+ = 2;$
$\}$
$closeUpCode();$
```

The SLMS version after decomposition is as follows.

```
$i = 0; j = 1;$
$startUpCode();$
$reg1 = a[i + 2];$
$a[i] = reg1;\ ||\ reg2 = a[j + 2];$
$while(a[j + 3]$ && $a[i + 3])\{$
   $i+ = 2;\ ||\ a[j] = reg2;\ ||\ reg1 = a[j + 3];$
   $j+ = 2;\ ||\ a[i] = reg1;\ ||\ reg2 = a[i + 3];$
$\}$
$closeUpCode();$
```

Note: this outcome is better (in terms of extracted parallelism) than the unrolled version.

The second extension is to apply SLMS to loops with conditional if-statements. The solution of section 3.1 using source level if-conversion is not very efficient as it adds conditional checks before every statement of the if-then/if-else body. Instead we can use the following idea (to the best of our knowledge a novel one but there is some similarity to the work of [20]):

- Let L be a simple loop with an if-statement $L = \ for(i = 0; i < n; i++)\{if(A_i)B_i\ else\ C_i;\ D_i;\}$.

- Assume that we can identify (via profile information or static analysis) that $Pf = A_i; B_i; D_i$ is the most frequent path.

- We can chose the II according to Pf assuming that it is executed repeatedly many times. By overlapping successive iterations of Pf a kernel $KPf = \; D_i||B_{i+1}||A_{i+2}$ may be obtained. Note that KPf can be repeatedly executed as long as A_{i+2} is evaluated to true.

- Thus, when A_{i+2} is false we must:

  - exit KPf.
  - drain the pipeline by executing $D_{i+1}; C_{i+2}; D[i+2]$.
  - continue executing the original loop until KPf can be re-started.

- Note that the less efficient fix-up code for draining the pipeline and locating a restart point for KPf is not executed frequently.

The process of transforming a loop with if-statements is schematically depicted in figure 23. This method can be generalized to loops with more than one if-statements and to loops with nested if-statements. The final code for the loop's kernel is a s follows.

```
$for(i = 0; i < n - 2; i + +)\{$
  $D[1];\ B[i + 1];$
  $if(!A[i + 2])\{$
    $D[i + 1]; \quad C[i + 2]; \quad D[i + 2];$
    $for(i = i + 3; i < n; i + +)\{$
      $if(!A[i])\{\ C[i]; \quad D[i]; \}$
      $else\ \{$
        $B[i];$
        $if(a[i + 1])\{$
          $i - -;$
          $break;$
        $\}\ else\ \{$

          $D[i];$
        $\}\}\}\}\}$
```

11 Conclusions

In this work a method for source level modulo scheduling (SLMS) has been developed and implemented in the Tiny parallelizer. In spite of its relative simplicity it obtained good speedups over the GCC (with and without the Swing MS), ICC and XLC as-well improvements of power-dissipation on ARM. Experimental results show that SLMS can have a different effect depending on the compiler and architecture hence SLMS must be applied selectively. The bad cases of performance degradation can be attributed to the additional array references inserted by the SLMS transformation. It turned out that by applying SLMS to loops with more than six arithmetic

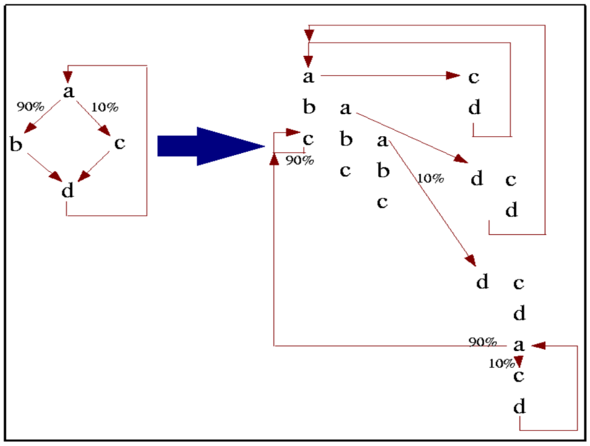


Figure 23: *Schematic representation of SLMS that is focused on the most frequent path.*

operations per each array references almost all of these bad cases can be eliminated.

To the best of our knowledge this is the first time SLMS has been demonstrated and implemented. This work, also presents two possible extensions to SLMS. An extended solution to loops with if-statements, and a partial solution to while-loops. The development of these extension will enable the application of SLMS to complex loops, thus allowing SLMS to use the full power of source level transformations. Register pressure (a critical issue with machine level MS) basically did not occurred in our experiments (except for kernel 10), in spite of the extensive parallelism obtained by the SLMS. This also may be attributed to the fact that register allocation and code generation are executed after SLMS.

The relation between SLMS and known loop transformations has been considered and demonstrated. The fact that SLMS is a source level optimization implies that it can be easily combined with other loop transformations to form a source level compiler (SLC) (a tool currently developed at Haifa

University). Though, the relation of SLCs and SLMS is not the focus of this work, it is an important usage of SLMS. Other compilers such as Polaris [7] that apply loop transformations and are able to generate C source code should not be considered as SLC as they produce C code from machine level intermediate representation. More related are real SLCs such as the LoopTool [15] interactively applying controlled loop fusion and unroll-and-jam to optimize programs at source level. Finally automatic parallelizers acting as a SLC such as the Parafrase system [13] can also benefit from using SLMS.

SLMS is useful for two tasks: as an addition to the arsenal of loop transformations for a source level compiler and as a preliminary optimization that differs from machine MS. We have proved, via examples and experiments that SLMS can lead to different scheduling results than machine level MS. Thus, SLMS can be also used as a regular optimization.

References

[1] Sim-panalyzer: http://www.eecs.umich.edu/panalyzer/.

[2] J. Ullman. A. Aho, R. Sethi. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[4] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[5] David Bailey. Nas kernel benchmark program: http://www.netlib.org/benchmark/nas.

[6] J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: Past, present, and future: http://www.netlib.org/utk/jackdongarra.

[7] K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua, and P. M. Petersen. The Polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, 1994.

[8] J. Huang and T. Leng. Generalized loop-unrolling: a method for program speed-up, 1997.

[9] Sverre Jarp. Optimizing IA-64 performance. *Journal of Software tools*, 26(7):21–22, 24, 26, July 2001.

[10] M. Lam. Software pipelining : an effective scheduling technique for vliw machines. In *PLDI*, pages 318–328, 1988.

[11] F. H. McMahon. Lawrence livermore national laboratory fortrn kernel:mflops.

[12] V. R. North. Ia-64 code generation: http://citeseer.ist.psu.edu/385244.html.

[13] C. D. Polychronopoulos, M. B. Gikar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. The structure of parafrase-2: an advanced parallelizing compiler for c and fortran. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 423–453, 1990.

[14] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[15] A. Qasem, G. Jin, and J. Mellor-Crummey. Improving performance with integrated program transformations. Technical Report TR03-419, Rice University, 2003.

[16] B. R. Rau and C. D. Glaese. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceeding of the 14th Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[17] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *MICRO*, pages 63–74, 1994.

[18] B. Ramakrishna Rau. Iterative-modulo-scheduling. In *HPL-94-115*, November 22,1995.

[19] Warter, Lavery, and Wwu (1993). The benefit of predicated execution for software pipelining. In *HICSS-26 Conference Proceedings*, page Vol. 1, January 1993.

[20] N. J. Warter, J. W. Bockhaus, G. E. Haab, and K. Subramanian. Enhanced modulo scheduling for loops with conditional branches. In *The 25th Annual International Symposium on Microarchitecture*, Portland, Oregon, 1992. ACM and IEEE.

[21] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. *SIGPLAN Not.*, 28(6):290–299, 1993.

[22] M. Wolfe. The tiny loop restructuring research tool. In *Proceedings of the International Conference on Parallel Processing*, 1991.

[23] A. M. Zaky. *Efficient Static Scheduling of Loops on Synchronous Multiprocessors.* PhD thesis, Ohio State University, OH, 1989.