Towards a Source Level Compiler: Source Level Modulo Scheduling

Yosi Ben-Asher Danny Meisler
Computer Sci. dep. Haifa University, Haifa.
Email:yosi/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 scheduling. 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 in 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) [11] in software level rather than implementing it in machine level, as is usually done in modern compilers [9]. 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" its iterations as follows:

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

Note that after this "pipelining" the dependence 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 loop iterations by eliminating the maximal number of inter iteration dependencies [2, 14].

This work considers another possibility of implementing MS, namely to 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 [3]. 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). We believe 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 his 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 the SLMS to use an Initiation Interval (II) [11] that violates some data dependencies. 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 in source level (as shown later on several possible combinations).

[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.

[copyright notice will appear here]

Figure 1 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 that can be obtained by using a machine level MS. Remark: some MS algorithms such as Iterative MS [12] 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.

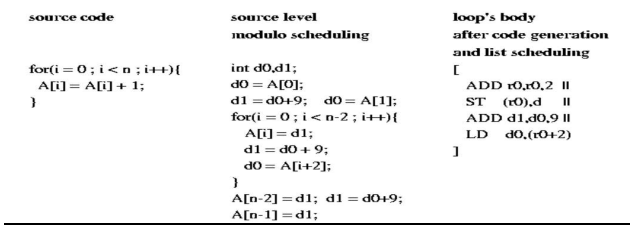

Figure 1. *Using SLMS followed by List scheduling.*

2. Basic operations used by the SLMS algorithm

In the following subsections we present 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). 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 2. 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 2 contains a single MI.

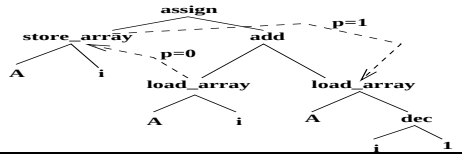

Figure 2. *Input structure for the SLMS algorithm.*

Next, we describe the concept of the minimum initiation interval (MII) [11] 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 [3]. 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 instructions are executed one after the other. For SLMS a different notion of delays will be defined as pipeline stalls has no meaning in 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 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.

Next, If-statements of the AST are eliminated by predicating them with Boolean variables, similar to the if-conversion operation performed in assembly mode, e.g., $if(x < y)\{x++;\ y--;\}$ is converted to two predicated MIs $c = (x < y);\ c?x++;\ !c?y--;$ so that SLMS can be applied. Remark: apart from the use of if-conversion in MS there are other proposals for MS of loops with conditional statements. For example, Lam [7] uses a sequence of hierarchical reductions of strongly connected components to MS a loop with conditional statements.

2.1 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 dependence 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 three additions. Assumption that the underlying CPU is a VLIW machine allowing up to two additions and two load/store operations 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 dependence prevents finding the MI (section 4).

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 dependence 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:

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

The data dependency of $reg1 = ...$ and $A[i-2] + reg1+$ will be eliminated by applying Modulo Variable Expansion (MVE), described in section 2.2. At this stage SLMS can be applied with

$II = 1$ as follows:

$$
\begin{array}{l}
reg1 = A[4];\\
for(i = 2; i < (N-3); i++)\{\\
\quad A[i] = A[i-1] + A[i-2] + A[i+1] + reg1; \;||\\
\quad reg1 = A[i+3];\\
\}\\
A[i] = A[i-1] + A[i-2] + A[i+1] + reg1;
\end{array}
$$

The symbol $||$ is used between multi intructions that can be totaly 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".

2.2 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 2.1. 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) [7] 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 dependence inside each unrolled copy of the kernel is removed.

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

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 by the $||$ symbol in each row).

The following example (Figure 3) 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.

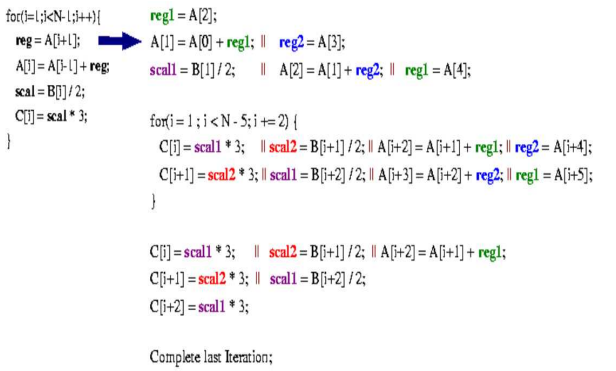

Figure 3. *SLMS decomposition and original loop scalar.*

[2] The number of times we need to unroll the loop depends on the lifetime of each variable in the loop as described in [7].

2.3 Scalar expansion

Another possibility to remove data dependencies caused by scalar variables is to use scalar expansion [3] and replace the scalar variable by a sequence of array references. For example, instead of applying MVE on the loop of section 2.1 scalar expansion can be applied by replacing $reg1$ by $regArr[i]$ so that the SLMS will be:

$$
\begin{array}{l}
regArr[2] = a[2];\\
for(i = 0; i < (N-2); i++)\{\\
\quad a[i] = a[i-1] + a[i-2] + a[i+1] + regArr[i+2];\\
\quad || \; regArr[i+3] = a[i+3];\\
\}\\
a[i] = a[i-1] + a[i-2] + a[i+1] + regArr[i+2];
\end{array}
$$

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

2.4 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 dependence edge (Figure 2) 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 dependence will be violated in the resulting kernel. Let MI_i, MI_j be two MIs connected by a dependence edge $e_{i,j}$ then the $delay(MI_i, MI_j)$ is defined as follows:

1. $delay(MI_i, MI_j) = 1$ if $i = j$ (self dependence).
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.
4. $delay(MI_i, MI_j) = 1$ if $e_{i,j}$ is a back edge.

Figure 4 depicts a data dependence 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 4), a feasible schedule is obtained for $MII = 2$ and not for $MII = 1$ which violates the backedge from f to c.

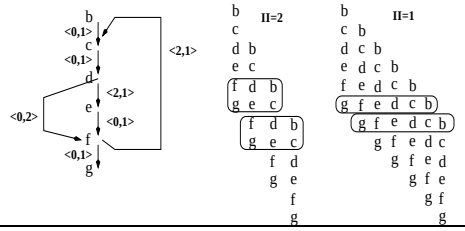

Figure 4. *delays between MIs.*

2.5 Computing the MII

In SLMS the MII accounts only for recurrence constraint (PMII [11]). 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 [2, 14] has been selected for two reasons.

1. First, its simplicity and its ability to naturally handle the case where each dependence 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.

3. 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 reduces 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. 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.

$$
\begin{array}{l}
for(k=0;k<n;k++)\{ \\
\quad CT = X[k,i]; \\
\quad X[k,i] = X[k,j] * 2; \\
\quad X[k,j] = CT; \\
\}
\end{array}
$$

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.

4. 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 3).
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 2.5).
   (b) Obtain the delays of the data dependencies edges (section 2.4).
   (c) Compute the MII (section 2.5).
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 2.1) based on data dependenc 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 2.2), save the maximum lifetime.
   (b) Build the prologue kernel and epilogue.
   (c) For each decomposed MI, MVE (section 2.2) or Scalar Expansion (section 2.3) is applied to eliminate dependencies caused by the decomposition. MVE or Scalar Expansion may also be activated 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 [2], 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 [14] 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.

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

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

$$
\begin{array}{l}
max = arr[0]; \\
for(i=0;i<n;i++) \\
\quad if(max < arr[i]) max = arr[i];
\end{array}
$$

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

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

Note: The last line was added manually.

5. SLMS and other loop transformations

SLMS can be combined with other loop reordering and restructuring transformations [3]. In 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 dependence of $a[i,j+1] = t$; and $t = a[i,j+1]$; as depicted by the following erroneous kernel obtained by using $II = 1$:

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

Using loop interchange [3] 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 separeted by $||$.

$$
\begin{array}{l}
for(j=0;j<n;j++)\{ \\
\quad for(i=0;i<n;i++)\{ \\
\quad\quad t\ =\ a[i,j]; \\
\quad\quad a[i,j+1] = t; \\
\quad \}
\end{array}
$$

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

is transformed to

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

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 [3]. Each of the following two loops can not be SLMSed due to the dependence 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:

$$
\begin{array}{l|l}
for(i=1;i<n;i++)\{ & \\
\quad t=A[i-1]; & \\
\quad B[i]=B[i]+t; & for(i=1;i<n;i++)\{ \\
\quad A[i]=t+B[i]; & \quad t=A[i-1]; \\
\quad \} & \quad B[i]=B[i]+t; \\
//second\ loop & \quad A[i]=t+B[i]; \\
for(i=1;i<n;i++)\{ & \quad q=C[i-1]; \\
\quad q=C[i-1]; & \quad B[i]=B[i]+q; \\
\quad B[i]=B[i]+q; & \quad C[i]=q*B[i]; \\
\quad C[i]=q*B[i]; & \} \\
\} & \\
TR &
\end{array}
$$

yielding

$$
\begin{array}{l}
t=A[i-1]; \\
B[i]=B[i]+t; \\
A[i]=t+B[i]; \\
q=C[i-1]; \quad ||t=A[i]; \\
B[i]=B[i]+q; ||B[i+1]=B[i+1]+t; \\
C[i]=q*B[i]; \;||A[i+1]=t+B[i+1]; \\
\qquad\qquad q=C[i]; \\
\qquad\qquad B[i+1]=B[i+1]+q; \\
\qquad\qquad C[i+1]=q*B[i+1];
\end{array}
$$

Consider two loops, applying SLMS 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 5 demonstrates this case.

SLMS can also be used to enable the application of loop transformations. For example, the following two loops (Figure 6) 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 6) 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.

6. Working with the source level compiler

In this section 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. First it is important to understand the difference between optimizing in source level mode and in machine level mode. In machine level the optimization can use exact

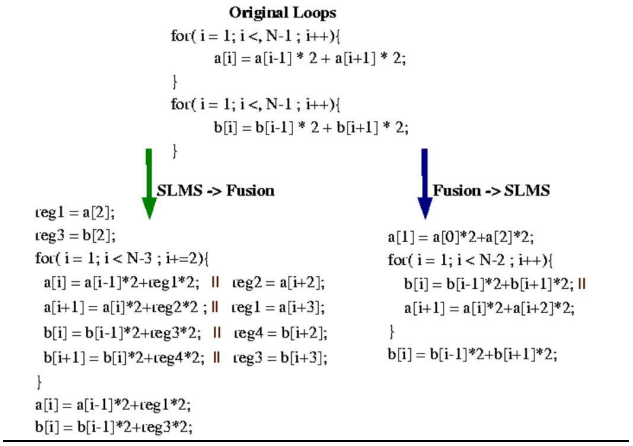

Figure 5. *The order of transformations changes the final scheduling.*

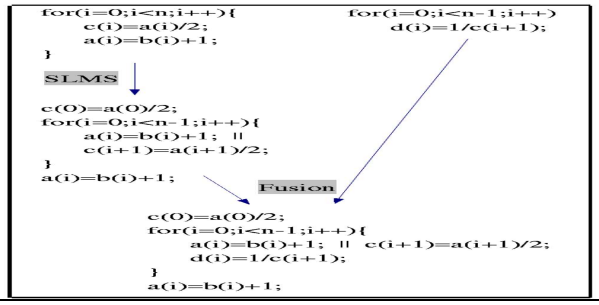

Figure 6. *SLMS allows loop fusion.*

knowledge of the CPU resources and obtained optimized scheduling. The opposite is true for source level optimization which is actually performed ignoring hardware resource constraints, optimizing for maximal parallelism at source level. This "disadvantage" can work to the benefit of a SLC. In particular, it can happen that due to hardware resource constraints the underlying MS will not optimize a given loop while after SLMS an optimized scheduling will be obtain. Typically, even an elementary list scheduling of basic blocks applied after SLMS can in some cases find better scheduling than the more constrained machine level MS.

As an example consider the loop $a[i] = a[i-2] + a[i+2];$ of figure 7 where the code generation used rotating registers [6] to create the loop's code. The underlying MS parallelizes the loop ($MII = RecII = 1$) due to the dependence cycle between the "load" and the "add". Note, that the "add" was assigned a delay of 2 cycles. The Data Dependecy (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 $a[i] = a[i-2] + reg;\ \ reg = a[i+3];$. 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 constraints).

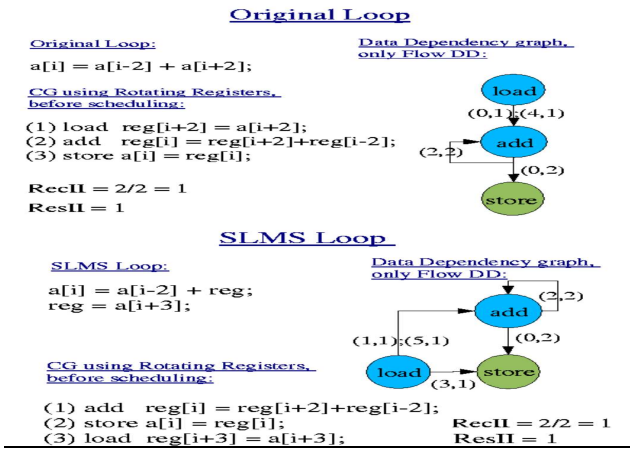

Figure 7. *SLMS changes the DD graph thus enabling other scheduling options.*

Apart from this ability of SLMS to find optimized scheduling by first ignoring resource constraints, there are some technical factors working in favor of SLMS. It is common that compilers restrict MS to small loops such as loops with less than 50 instructions. SLMS is significantly faster than machine level MS, as it does not have to schedule under detailed resource constraints. In addition SLMS works in source level thus can naturally determine the exact dependencies between each two array references. Though a compiler can also obtain these dependencies in 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 $A[2*(i+1)]$ are more complicated to implement in RTL/machine level than in source level.

7. Experimental results

SLMS was implemented in Wolfe's Tiny system [13] enhanced by the Omega test [10]. 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 [4] benchmark, Livermore [8] loops, Linpack [5] 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 intruction level parallelism.

7.1 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 8, 9 and 10, present speedups obtained using GCC (IA64) over ItaniumII 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 in 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 dependence 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 dependence between MI and revealed the perellelism between them, thus enabling the generation of less bundles. Indeed beffore 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 favour 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 in source level by general high level characteristics, experimental results prove that they are specific for the pair compiler/hardware.

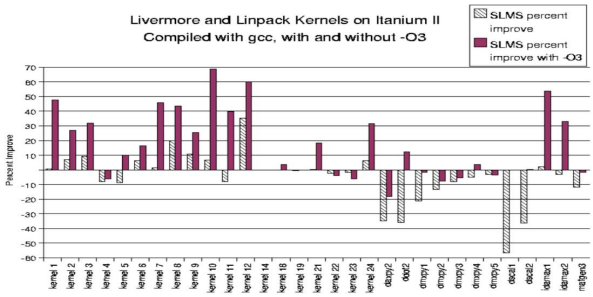

Figure 8. *Livermore & Linpack over GCC*

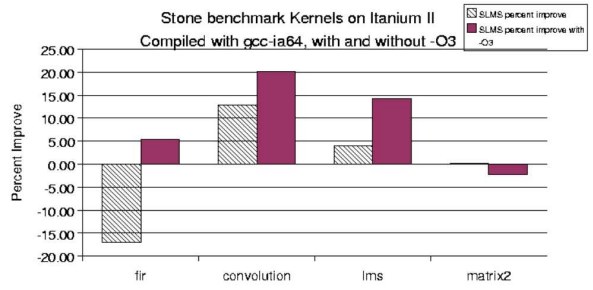

Figure 9. *Stone over GCC*

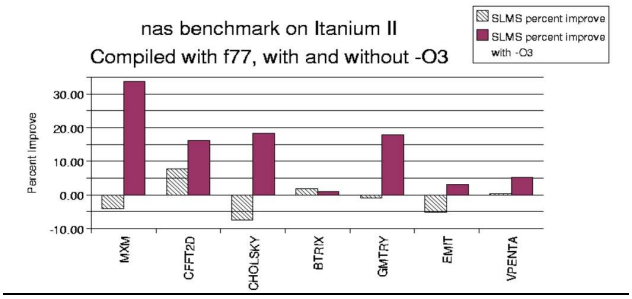

Figure 10. *NAS over GCC*

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 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 11 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.

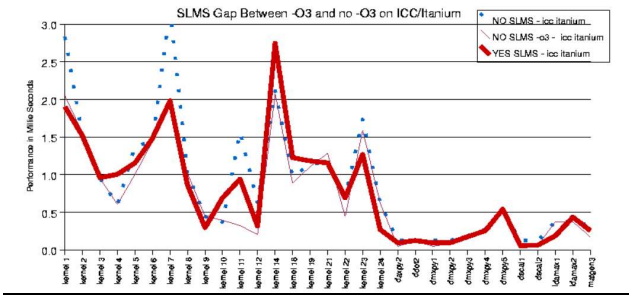

Figure 11. *SLMS can be used to close the gap between using and not using -O3.*

We also tested SLMS on a superscalar processor (Pentium(R)) where all the parallelism is obtained by the HW pipeline. Figure 12 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.

7.2 Experimental results over highly optimizing compilers

The following graphs 13, 14 15, 16 and 17, 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 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.

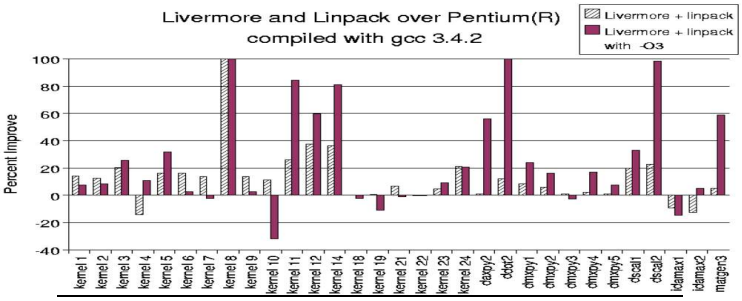

Figure 12. *SLMS can improve performance over superscalar processor.*

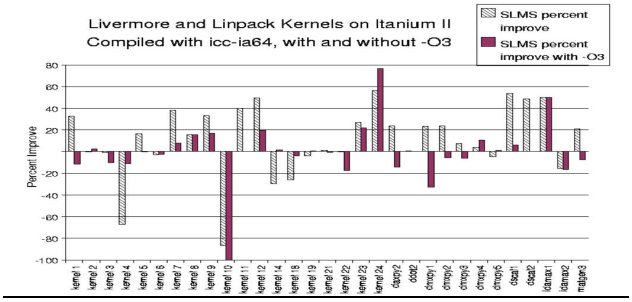

Figure 13. *Livermore & Linpack over ICC*

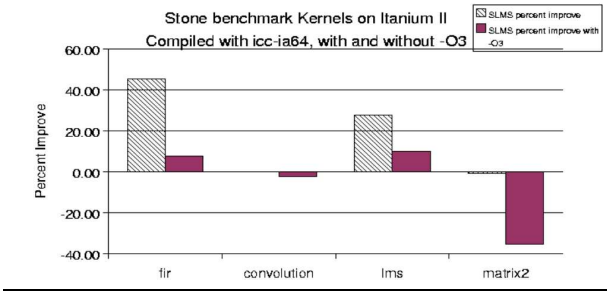

Figure 14. *Stone over ICC*

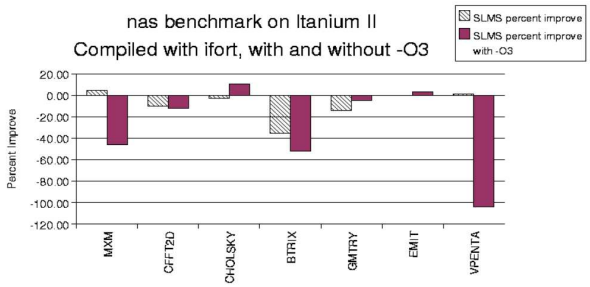

Figure 15. *NAS over ICC*

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

$$
\begin{array}{l}
float\ k[n];\\
for(k=1;k<n;k++)\\
\{\\
\quad X[k]=X[k-1]*X[k-1]*X[k-1]*X[k-1]*X[k-1]+\\
\qquad X[k+1]*X[k+1]*X[k+1]*X[k+1]*X[k+1];\\
\}
\end{array}
$$

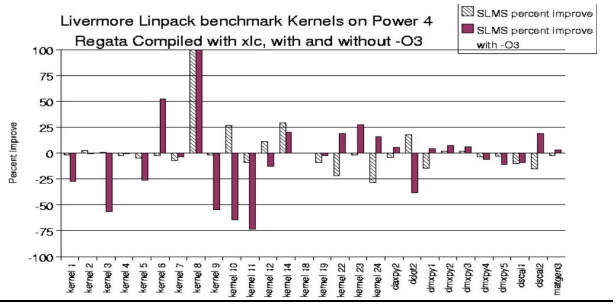

Figure 16. *Livermore & Linpack over XLC*

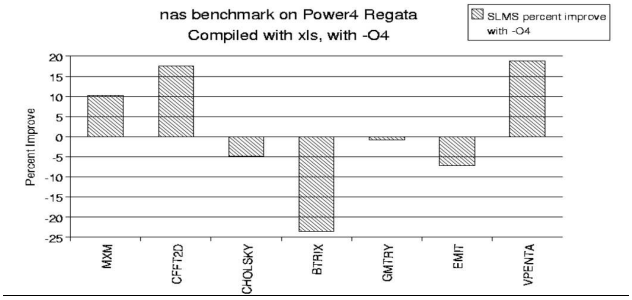

Figure 17. *NAS over XLC*

7.3 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 18 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. 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 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 18 should be regarded as a success, provided that SLMS will be used selectively.

8. 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. To the best of our knowledge this is the first time SLMS has been demonstrated and implemented. We have used a simple version of

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

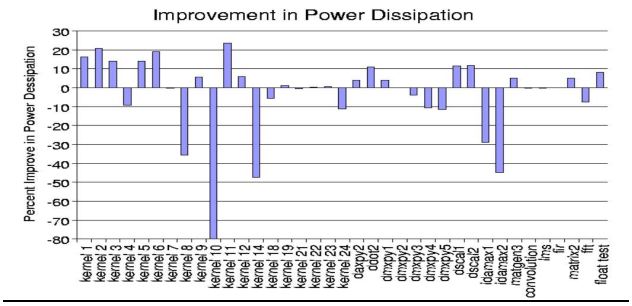

Figure 18. *Power dissipation for the ARM*

if-conversions for loops with if-statements. However, a more aggressive type of solution is possible (not included here) allowing the SLMS to use the full power of source level transformations. Also, partial solutions to while-loops are also possible using the simplicity of source level mode of work. 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. SLMS is useful for two tasks: an addition to the arsenal of loop transformations for a source level compiler and as a preliminary optimization 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] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

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

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

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

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

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

[8] F. H. McMahon. Lawrence livermore national laboratory fortrn kernel:mfbps.

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

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

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

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

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

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