

Reaching Fast Code Faster: Using Modeling for Efficient Software Thread Integration on a VLIW DSP *

Won So and Alexander G. Dean
Center for Embedded Systems Research
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695-7256
{wso, alex_dean}@ncsu.edu

ABSTRACT

When integrating software threads together to boost performance on a processor with instruction-level parallel processing support, it is rarely clear which code regions should be aligned and integrated, and which regions should be left alone. This problem grows even worse on a modern VLIW DSP due to complicating factors in both the hardware and compiler: software pipelining, predication, branch delay slots, load delay slots and limited resources. As a result, finding an effective integration strategy requires extensive iteration through the integrate/compile/analyze sequence.

In this paper we introduce methods to quantitatively estimate the performance benefit from the integration of multiple software threads. We use resource modeling, consider register pressure and compensate for compiler optimizations. This enables different scenarios to be compared and ranked. We then use these estimates to guide integration by concentrating on the most beneficial scenario. Information from each iteration of compilation is used to update the rankings of scenarios. We find that our modeling methods combined with limited compilation quickly find the best integration scenario without requiring exhaustive integration.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—code generation, compilers, optimization

General Terms

Algorithms, Experimentation, Design, Performance

Keywords

Software thread integration, static profitability estimation, iterative compilation, software pipelining, VLIW, DSP, TI C6000

*This material is based upon work supported by NSF CAREER award CCR-0133690.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES'06, October 23–25, 2006, Seoul, Korea.

Copyright 2006 ACM 1-59593-543-6/06/0010 ...\$5.00.

1. INTRODUCTION

Many modern microprocessors and digital signal processors (DSPs) are capable of issuing multiple instructions per cycle. However, they typically fall short of full utilization due to a lack of independent instructions. Software pipelining (SWP) is a critical optimization which improves loop performance on such processors. Hence any further optimizations must coexist with SWP.

Existing methods for software thread integration (STI) for very long instruction word (VLIW) processors [28] address how to complement and reinforce software pipelining by selectively applying STI based upon characteristics of two loops. Each loop's assembly code is examined to characterize the loop based upon speedup from SWP and whether dependence bounds and complex control flow limited SWP's effectiveness. A transformation rule matrix is examined based upon characteristics of the two loops to determine whether to integrate, and if so, how. These existing methods use a general test which provides a yes/no answer to whether to integrate, and high-level information on which methods to use. However, no attempt is made to quantify the performance of the integrated code, limiting the ability to objectively rank potential combinations of loops.

The first contribution of this paper is methods to quantitatively estimate the performance impact of integration, allowing various integration scenarios to be compared and ranked, leading to more efficient integration. This is a difficult problem because of complexity in both the hardware and the compiler for modern processors (the Texas Instruments TMS320C64x DSP in this case). Software pipelining, predication, branch and load delay slots and limited resources all complicate the prediction of the impact of integration and software pipelining. We rely on resource modeling, emulating list scheduling (via *pseudo-scheduling*) and take into account register pressure and machine features such as predication, and compensate for compiler optimizations as needed.

The second contribution of this paper is methods which use these quantitative estimates to guide the alignment of regions in two procedures to yield the fastest code with a small number of iterative compilation cycles. This iterative approach allows inaccuracies in the static performance estimation to be corrected quickly. We demonstrate the methods using code examples from the Texas Instruments DSP and image processing libraries and MiBench. We find that our approach effectively identifies good integration strategies.

This paper is organized as follows. Section 2 describes the methods for analysis, estimation, code transformation and iterative compilation. Section 3 presents the hardware and software characteris-

tics of the experiments run, which are analyzed in Section 4. Section 5 summarizes the related work.

2. METHODS

Software thread integration (STI) is essentially procedure jamming (or fusion) with intraprocedural code motion transformations which allow arbitrary alignment of instructions or code regions. This alignment allows code to be moved to use available execution resources better and improve the execution schedule. In our previous work [27], we investigated how to select and integrate procedures to enable conversion of coarse-grain parallelism (between procedures) to a fine-grain level (within a single procedure) using *procedure cloning and integration*.

For this paper, we focus on an efficient method for finding the best integrated procedure from two independent procedures when there are multiple possible scenarios. The proposed method addresses *the problem of integrating two procedures with arbitrary control flow and utilization*. Though dealing with 3 or more procedures including multiple call hierarchies is beyond this work, this is a key step for extending STI to more complicated cases.

Integration begins with an appropriate representation of original procedures. Analysis of the compiled code provides useful information to identify potentially profitable code combinations. After defining the design space for possible integration scenarios, we develop an efficient searching technique to find the best scenario by incorporating static estimation with iterative compilation.

2.1 Program Representation

STI uses the control dependence graph (CDG, a subset of the program dependence graph) to represent the structure of the program: its hierarchical form simplifies analysis and transformation. CDGs are constructed for both host and guest (or secondary and primary) threads to move code from the guest thread to host thread. Since the STI code transformations have been traditionally done at the assembly language level, the CDGs have been constructed from (compiler-generated) assembly code.

For this work, we use transformations at the C source level to avoid duplicating the complexity of a VLIW compiler [27, 28]. For transformation consistency, the CDG is constructed from the source code. In order to do this, we modify some definitions in the original CDG and redefine the graph as the *Simplified CDG (SCDG)*. We limit our scope to structured source code because most C code is structured and most non-structured code can be converted into structured code. The rules to construct the SCDG follow:

1. A list of C statements without control flow change forms a *code* node. A subroutine call is not considered to be control flow change.
2. A loop statement (for, while, do-while) forms a *loop* node and a child node containing statements inside a loop body. The loop type, loop-entry condition and loop incremental expression are stored as attributes of a loop node, if any exist.
3. A conditional statement (if, if-else, switch-case) forms a *predicate* node and child nodes, each of which contains conditionally executed statements. The conditional type and condition expressions are stored as attributes of a predicate node.
4. The graph is constructed hierarchically by applying these rules repeatedly.

Figure 1 shows the example C source code and the corresponding SCDG. The S1,S2,...,S6 labels can represent a list of multiple statements as well as a single statement. The leaf code nodes

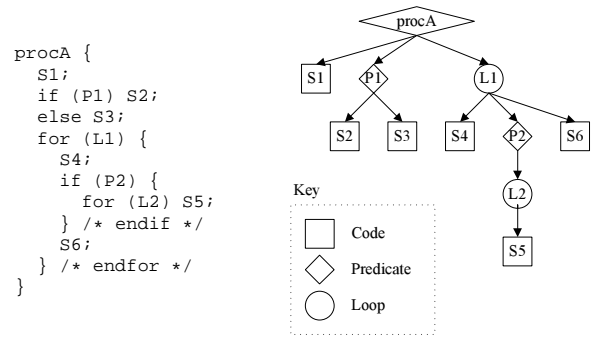


Figure 1: Example C source code and corresponding SCDG

which contain statements identify the code regions which can be integrated. Though predicate and loop nodes have corresponding C source expressions (P1, P2, L1 and L2), the impact of this code is mostly negligible because it includes considerably less source code than code nodes do, and we therefore do not expand these expressions.

2.2 Schedule Analysis

By analyzing the assembly code obtained after compilation, we extract useful information for efficient integration. This information is annotated into the SCDG and guides the following steps.

2.2.1 Schedule and profile information

After the original procedures are compiled separately, the assembly code for both procedures is obtained. For analysis, the instructions are divided into basic blocks (BB). We do not treat a subroutine call as a branch instruction for consistency with the previous step. By analyzing assembly code for each basic block, the following information is obtained:

1. Basic schedule information per basic block: number of scheduled cycles (i.e. schedule cycle count) and number of instructions
2. Software pipelining (SWP) information for every inner-most loop: Resource Minimum Initiation Interval (ResMII) and Recurrence Minimum Initiation Interval (RecMII), reason of SWP failure if SWP fails
3. Resource usage per cycle: instruction and its resource usage
4. Register usage per basic block: registers accessed regardless of use and definition

It is possible to construct a control flow graph (CFG) from the assembly code. Based on the CFG and the schedule cycle count of each BB, the static execution time (SET) of each procedure is computed. The SET of sequence of BBs is sum of their schedule cycle counts. If a BB is inside a loop, the SET is the product of its schedule cycle count and loop count. If a BB is conditionally executed, the SET is the product of the schedule cycle count and branch fraction. If it contains a subroutine call, we ignore cycles spent on the callee. Therefore, the SET of the whole procedure is summarized by the following equation:

$$SET_{procA} = \sum_{bb_i \in BB} s_{bb_i} f_{bb_i} n_{bb_i} \quad (1)$$

where BB is a set containing all basic blocks of the procedure, bb_i is an element in the set BB and s_{bb_i} , f_{bb_i} , n_{bb_i} represent the sched-

ule cycle count, the execution fraction and the execution count of the bb_i respectively.

Since the execution count (n) and the execution fraction (f) are determined at run time in most cases, the execution profile is necessary to compute the SET. If the given profile is completely consistent with the actual run, the SET represents the actual run time. A discrepancy between the profile and the actual run may cause the SET and actual run time to differ. Besides the SET, the following steps are affected because they also use the profile to compute profitability. However, it is a common weakness of profile guided optimizations rather than one specific to this method.

If the execution profile is not provided, we can generate it based on static prediction or estimation schemes. For example, we can treat every if-else conditional body equally by setting f to 0.5. Looping code can have higher impact than the other code by fixing n to a certain number, which is considered as priority of loops over acyclic code. The following steps guided by this profile lead to the improvement of the average case instead of the specific case of the provided execution profile.

2.2.2 Modification and annotation of SCDG

In order to use the information from schedule analysis and profiling to guide integration, the information must be annotated to the SCDGs of both procedures. Since the SCDG is constructed from the C source code while the information for each basic block is extracted from the compiled assembly code, the SCDG must be modified to reflect possible control flow changes caused by compiler optimizations.

The first source of this change is the compiler’s front-end optimizations. The compiler may unroll loops with a fixed number of iterations and a small loop body. If loop unrolling is applied, the SCDG is modified to match with the compiled code. If any other front-end optimizations are performed, the appropriate transformation is followed. The second source is predication. The compiler tries predication for conditionals if there is architectural support. If conditionals are predicated, they no longer cause a control flow change, hence they are treated as one statement during code transformation. This causes the predicate node and its child nodes to be merged into the parent code node or constructed as a separate code node.

Once the control flow of the SCDG and the assembly code are consistent, we annotate each code node in the SCDG with the schedule and profile information given by the previous step. Matching a code node in the SCDG with a BB in the compiled assembly code is often nontrivial because the assembly code has undergone numerous optimization phases. The compiler optimizations can change control flow as a byproduct of optimization. Instruction scheduling techniques can move instructions beyond basic block boundaries so that the boundaries between statements in the source code become ambiguous after the code is scheduled. Hence, there is no ‘perfect’ one-to-one mapping between code nodes in the SCDG and BBs in the assembly code.

The ‘best’ mapping is pursued by choosing the BB which has the most influence in terms of performance. For a code node under a loop node, only the loop body is considered. When a loop is software pipelined, the SWP kernel is used while the prolog and epilog are ignored. For a code node under a predicate node, the BB with the most instructions which correspond to the code node is selected. Remaining code nodes are matched with the rest of BBs in the assembly code by choosing in-between BBs.

2.3 Definition of Alignment Set

There are numerous possible integration scenarios depending on number of code nodes and their control and data dependencies. For efficient exploration, we first define the design space for this problem. Each integration scenario corresponds to a certain *alignment* of code nodes from two procedures, which determines which nodes are integrated with each other and which nodes are left unchanged. A *combination* indicates a pair of code nodes integrated with each other. Therefore, an *alignment* includes at least a single *combination* of code nodes. Let R_A and R_B denote the code node sets for procedure A and B respectively.

$$R_A = \{a_1, a_2, \dots, a_m\}, \quad R_B = \{b_1, b_2, \dots, b_n\}$$

where a_1, \dots, a_m and b_1, \dots, b_n represent the code nodes in procedure A and B respectively by the order in each SCDG. Let C_{AB} denote the set including all *combinations*, and L_{AB} denote the set including all *alignments* for procedure A and B.

$$\begin{aligned} C_{AB} &= \{(a_1, b_1), (a_1, b_2), \dots, (a_m, b_n)\} = R_A \times R_B \\ L_{AB} &= \{\{(a_1, b_1)\}, \{(a_1, b_1), (a_2, b_2)\}, \dots, \{(a_m, b_n)\}\} \\ &= \{x | x \subset C_{AB} \text{ and } x \neq \emptyset \text{ and } x \text{ is legal.}\} \end{aligned}$$

The combination set C_{AB} is the product set of R_A and R_B . Each element in the alignment set L_{AB} is a subset of C_{AB} which is *legal*. Being legal means that two distinct combinations in one alignment do not violate data dependencies of the original code. If we conservatively assume that succeeding statements in the original code are always data-dependent on preceding statements, it can be rephrased into these two conditions: two distinct combinations in one alignment (1) neither include the same code node (2) nor change the original order of the code nodes. For example, $\{(a_1, b_1), (a_1, b_2)\}$ is not legal because two combinations include the same code a_1 . The code a_1 can not be divided into b_1 and b_2 arbitrarily due to possible data dependency and control flow difference between them. The subset $\{(a_1, b_2), (a_2, b_1)\}$ is also not legal because the order of a_1 and a_2 forces b_1 and b_2 to be reversed. If a control dependency exists between code nodes from the same procedure, the conditions become more restrictive. In any case, legality is determined by checking if an alignment reverses the order of original code nodes. These conservative assumptions could be relaxed using data flow analysis but we leave this for future work.

Though STI code transformation enables arbitrary combinations of code nodes, allowing all possible combinations results in explosion of the alignment set size. The number of combinations increases quadratically (i.e. $n(C_{AB}) = n(R_A) \times n(R_B) = mn$) and the upper bound of the number of alignments increases exponentially (i.e. $mn \leq n(L_{AB}) < 2^{mn}$) with the number of code nodes.

In order to reduce the number of alignments, we include only combinations of code nodes under loop nodes (i.e. cyclic code) in the alignment set, as this code dominates performance. The combinations of other code nodes which are not under loop nodes (i.e. acyclic code) are examined after finding the best alignment of cyclic code. Second, we exclude combinations which have an extremely low probability of speedup such as SWP and Non-SWP loop combinations.

2.4 Static Profitability Estimation

The alignment set given by the previous step defines the design space for integration. If the size of the alignment set is small, it is not hard to examine all possible scenarios by compiling all integrated procedures. However, it is quite ineffective if there are nu-

merous possible alignments. Therefore, it is necessary to prioritize alignments based on an integration profitability measure.

Static profitability estimation is a modeling method for estimating profitability of a certain alignment. Since we are interested in improving run-time performance by integration, the appropriate measure of profitability is execution cycle count reduction. For modeling, we assume the following: (1) Only the parts of code which have changed after integration (i.e. integrated code nodes) influence the execution cycle count. (2) Overall profitability of a certain alignment is sum of profitability of code combinations included in the alignment. If the alignment $l_i \in L_{AB}$ includes multiple combinations c_1, \dots, c_n (i.e. $l_i = \{c_1, \dots, c_n\}$), the *estimated profitability (EP)* for this alignment is given by sum of the profitability of each combination:

$$EP_i = \sum_{c_k \in l_i} p_{c_k} \quad (2)$$

The profitability of a certain code combination (p_{c_k}) depends on two factors: (1) The schedule cycle count reduction by integration of code nodes (Δs_{c_k}) and (2) the impact of the integrated code on execution cycle count. The first is determined by how the compiler schedules the integrated code and the second is the product of the execution fraction (f_{c_k}) and the execution count (n_{c_k}). If $a_i \in R_A, b_j \in R_B$ denote code nodes to be integrated and $c_k = (a_i, b_j) \in C_{AB}$ denote the combination of those, the estimated profitability of the combination c_k (p_{c_k}) is computed by the following equations:

$$f_{c_k} = f_{a_i} f_{b_j} \quad (3)$$

$$n_{c_k} = \min(n_{a_i}, n_{b_j}) \quad (4)$$

$$\Delta s_{c_k} = s_{a_i} + s_{b_j} - s_{c_k} \quad (5)$$

$$p_{c_k} = \Delta s_{c_k} f_{c_k} n_{c_k} \quad (6)$$

where $f_{a_i}, f_{b_j}, f_{c_k}$ are the execution fractions, $n_{a_i}, n_{b_j}, n_{c_k}$ are the execution counts, and $s_{a_i}, s_{b_j}, s_{c_k}$ are the schedule cycle counts of a_i, b_j, c_k respectively.

Since the schedule (s_{a_i}, s_{b_j}) and profile ($f_{a_i}, f_{b_j}, n_{a_i}, n_{b_j}$) information are already provided, the only unknown variable is the schedule cycle count of the combination (s_{c_k}). The schedule of integrated code depends on various characteristics of two code nodes such as utilization, used registers and their usage patterns, data dependencies between instructions and so on. Since it is almost impossible to predict the accurate schedule without performing scheduling, we propose a method called *pseudo-scheduling* to estimate the schedule cycle count of the integrated code.

2.4.1 Pseudo-scheduling

Since VLIW machines use static instruction scheduling, the schedule is embedded in the compiled code. We convert the instruction schedule of each BB into the *resource map (RM)*, a table where number of rows is the schedule cycle count and number of columns is the number of functional units (FU). For each instruction in a BB, there is a mapping between the issued cycle and the issued FU. According to this mapping, each instruction is assigned to a certain entry of the table. Therefore, free spaces in the RM represents idle issue slots in the resulting schedule.

Given resource maps from two code nodes, pseudo-scheduling constructs the schedule of integrated code by moving instructions in one resource map to the idle slots in the other one. Two restrictions are applied during this process. The first one is resource compatibility. In most architectures, instruction and FU types are tied. Depending on its type, an instruction can be issued only to a certain FU or subset of FUs. The second restriction is the data

dependencies between instructions. We force instructions to maintain the same cycle distance between them after they are moved to the other resource map. We conservatively apply this restriction even though some instructions are independent. Dataflow analysis would loosen this constraint.

For software pipelined (SWP) loops, the method above does not make sense because instructions from different iterations are overlapped in the SWP kernel. Therefore, we use a specialized pseudo-scheduling method for SWP loops based on modulo scheduling algorithm. In modulo scheduling, the schedule cycle count of a SWP loop kernel (*Initiation Interval (II)*) is constrained by the *Minimum II (MII)*. The MII is the maximum of *Resource MII (ResMII)* and *Recurrence MII (RecMII)*, where the RecMII is determined by the length of recurring dependency and the ResMII is a function of the number of instructions which uses each resource. For ResMII, we use a *Resource Vector (RV)*, where each element is the number of instructions which uses a specific resource. The number of elements in a RV and the function for computing ResMII (*ComputeResMII*) are architecture dependent. For a TI C64x DSP, there are 20 types of resources involved in ResMII.

Besides ResMII and RecMII, the final II depends on the complexity of instruction sequence because the II is determined after the compiler finds a valid schedule. While II is equal to MII when the compiler find a schedule on the first try, II becomes larger than MII as it experiences more trouble to find one. The difference between II and MII (i.e. II - MII) grows with the complexity of the instruction sequence, which generally rises with more instructions, more dependencies between instructions, and more register pressure.

The following equations summarize the pseudo-scheduling method for software pipelined loops to estimate the schedule cycle count of the integrated code (s_{c_k}).

$$ResMII_{c_k} = ComputeResMII(RV_{c_k})$$

$$where \quad RV_{c_k} = RV_{a_i} + RV_{b_j}$$

$$RecMII_{c_k} = Max(RecMII_{a_i}, RecMII_{b_j})$$

$$MII_{c_k} = Max(ResMII_{c_k}, RecMII_{c_k})$$

$$s_{c_k} = II_{c_k}$$

$$= MII_{c_k} + (II_{a_i} - MII_{a_i}) + (II_{b_j} - MII_{b_j})$$

2.4.2 Register pressure estimation

Since pseudo-scheduling moves instructions from one resource map to the other without considering additional register pressure caused by integration, it ignores the possibility that the compiler may generate different or more instructions to resolve the register conflict. Hence the schedule obtained by pseudo-scheduling is usually better than the real schedule. If many additional spills occur due to heavy register pressure, the schedule of integrated code is much worse than that of original code. Therefore, it is desirable to avoid code combinations which cause spills. We use a model called *register pressure estimation*. We compute number of registers accessed by code nodes and estimate the register pressure of integrated code by adding them. If the sum is larger than the number of total physical registers, we discard it from the combination set. If there are predicated instructions, the predicate register pressure is estimated in the same way.

There are still some cases where differences between the estimated schedule and the real schedule are large. If the compiler generates different sequences of instructions for the original and integrated code, pseudo-scheduling can not be accurate because it assumes that the same sequence of instructions are generated for the same code after integration. For example, if the compiler suc-

ceeds in SIMD optimization for the integrated code though it fails for original code, the generated instructions show a big difference. However, it is extremely hard to predict how every optimization impacts code generation. In reality, instruction scheduling and register allocation are closely coupled, though we use separate models for those. These facts force estimation inaccuracy.

Though there are numerous options to improve the accuracy of estimation, we maintain this level of complexity and depend on the target compiler’s result to determine the actual profitability. There is a design trade-off between estimation and compilation. If estimation is as accurate as compilation, the best alignment is chosen based on this estimation without iterative compilation. However, the high computation complexity raises time spent on estimation significantly. For this work, the balance between moderate estimation complexity and target compiler’s feedback is pursued because we try to define a more general method than one tightly coupled with a certain architecture and compiler.

2.5 Code Transformation

Code transformation is the step where an alignment is converted to the corresponding integrated procedure. Since the alignment determines which parts of the code are integrated, different transformation techniques are applied depending on their control dependencies. Figure 2 illustrates basic code transformation techniques and corresponding estimated profitability (EP) for STI depending on the parent node types.

In addition to these methods, there are some pretransformation techniques which can be applied before integration. Those include loop unrolling and loop peeling as shown in Figure 3. Loop unrolling can be used to provide additional instructions combined with loop jamming. It is powerful in cases where two loop bodies are asymmetric in terms of size, utilization and loop counts [28]. Loop peeling can be used to lower a loop nesting level. If loops with different nesting levels need to be integrated, it is required to make both nesting levels identical by lowering the nesting level of one loop. Since only inner-most loops are software pipelined, it is often required to integrate SWP loops with different nesting levels. By combining and hierarchically applying both sets of techniques, any alignments can be successfully realized.

The transformation techniques presented above are targeted for the general case. Depending on the situation, some code can be removed if unnecessary. For example, if both loop counts are fixed and the same, copies of the original loops in Figure 2(a) are not necessary. Transformation leaves opportunities for further optimizations such as induction variable elimination. The jammed loops may have two induction variables unnecessarily after integration, which can be reduced into one.

2.6 Iterative Compilation

We depend on the final step of *iterative compilation* to determine which alignment is best. The method for iterative compilation presented here is a heuristic approach to find the best alignment with the least compilations (i.e. cost). We define the following distinct measures for an alignment:

1. Estimated profitability of the alignment l_i (EP_i) is obtained by the equation (2) and (6). Initial s_{c_k} is computed by pseudo-scheduling but is updated by the real schedule cycle count obtained after each compilation. Therefore EP_i converges to XP_i .
2. Expected profitability of the alignment l_i (XP_i) is computed by the same way as EP_i but uses s_{c_k} given by the real schedule of the integrated code after compilation.

- 1: Choose $l_i \in L_{AB}$ where $EP_i = \text{Max}(EP_1, \dots, EP_n)$;
- 2: **loop**
- 3: Perform code transformation for l_i ;
- 4: Compile integrated procedure;
- 5: Compute XP_i and AP_i ;
- 6: Update EP_k for all $l_k \in L_{AB}$ where $l_i \cap l_k \neq \emptyset$;
- 7: Remove all l_k where $EP_k < 0$ for all $l_k \in L_{AB}$;
- 8: Choose $l_j \in L_{AB}$ where $EP_j = \text{Max}(EP_1, \dots, EP_{n'})$;
- 9: **if** $j = i$ **then**
- 10: **break**;
- 11: **end if**
- 12: $i \leftarrow j$;
- 13: **end loop**
- 14: Choose $l_i \in L_{AB}$ where $AP_i = \text{Max}(AP_1, \dots, AP_{n'})$;

Figure 4: Algorithm for iterative compilation

3. Actual profitability of the alignment l_i (AP_i) is the static execution time (SET) difference between original procedures and the integrated procedure. The SETs are obtained by analyzing its assembly code as described in Section 2.2.

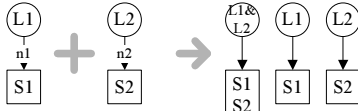
The EP_i is the only measure which can be computed initially before any compilation. The XP_i reflects the schedule of the integrated code but still implies some inaccuracy from side effects of code transformation. The AP_i reflects the real schedule of the whole integrated procedure. The inequality $EP_i \geq XP_i \geq AP_i$ therefore holds for most cases because the schedule from pseudo-scheduling is more optimistic than the real schedule and side effects of code transformation usually have a negative impact. The actual profitability of the integrated procedure is determined by AP_i while we use EP_i for prioritizing alignments and XP_i for determining when the iterative compilation cycle stops.

Figure 4 shows the algorithm for iterative compilation. First, the alignment (l_i) with the largest EP is chosen and an integrated procedure is generated (Line 1-4). After compilation, the schedule of integrated code is determined from the assembly code and the XP and AP are computed (Line 5). Since the real schedules of the code combinations (i.e. integrated code) included in the alignment are obtained, the EPs of l_i and other alignments which share the same combinations are updated (Line 6). After this update, the alignments with a negative EP are removed from the alignment set because compilation may have identified unprofitable code combinations (Line 7). Then we choose the alignment with the maximum of updated EPs again (Line 8). If the alignment is the same as the one just compiled, the iterative cycle stops (Line 9-10). Otherwise, another compilation cycle is performed with the newly chosen alignment (Line 12). Once the cycle stops, we choose the best integrated procedure from compiled ones based on the AP (Line 14).

This algorithm has two important implications. The first is a *feedback mechanism* implemented by an EP update. Since this feedback reprioritizes alignments based on updated EPs, it minimizes the impact of pseudo-scheduling inaccuracy on future alignments by considering previous compilation results. The second is a *decision mechanism* implemented by a comparison of EP and XP. After an EP update, EP is the same as XP for alignments which have been compiled. By comparing the EP(=XP) of compiled alignments with the EP of yet-to-be-compiled alignments, the algorithm effectively filters out the alignments which are not worthwhile because the EP generally determines the upper bound of the XP.

```
Code 1:
for(i=0;i<n1;i++) S1;
Code 2:
for(j=0;j<n2;j++) S2;
Integrated code:
for(i=0,j=0;i<n1&& j<n2;
    i++,j++)
    {S1; S2;}
for(i<n1;i++) S1;
for(j<n2;j++) S2;
```

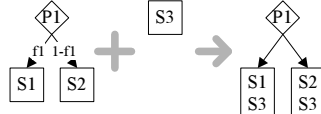
$$EP = \Delta_{s(S1,S2)} \times \min(n1, n2)$$



(a) loop + loop

```
Code 1:
if (p1) S1;
else S2;
Code 2:
S3;
Integrated code:
if (p1) {S1; S3;}
else {S2; S3;}
```

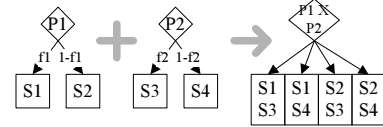
$$EP = \Delta_{s(S1,S3)} \times f1 + \Delta_{s(S2,S3)} \times (1-f1)$$



(b) predicate + code

```
Code 1:
if (p1) S1;
else S2;
Code 2:
if (p2) S3;
else S4;
Integrated code:
switch ( ((p1!=0)<<1) | (p2!=0) ) {
case 3: S1; S3; break;
case 2: S1; S4; break;
case 1: S2; S3; break;
default: S2; S4;
}
```

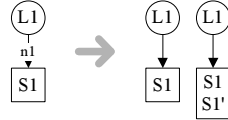
$$EP = \Delta_{s(S1,S3)} \times f1 \times f2 + \Delta_{s(S1,S4)} \times f1 \times (1-f2) + \Delta_{s(S2,S3)} \times (1-f1) \times f2 + \Delta_{s(S2,S4)} \times (1-f1) \times (1-f2)$$



(c) predicate + predicate

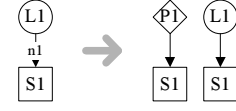
Figure 2: Code transformation techniques for STI

```
Before:
for(i=0;i<n1;i++) S1;
After:
for(i=0;i<n%2;i++) S1;
for(i<n;i+=2) {S1;S1';}
```



(a) loop unrolling

```
Before:
for(i=0;i<n1;i++) S1;
After:
if(n1>=1) S1;
for(i=1;i<n1;i++) S1;
```



(b) loop peeling

Figure 3: Pretransformation techniques for STI

3. EXPERIMENTS

3.1 Target Architecture

Our target architecture is the Texas Instruments TMS320C64x. From TI's high-performance C6000 VLIW DSP family, the C64x is a fixed-point DSP architecture with extremely high performance. The processor core is divided into two clusters with 4 functional units and 32 registers each. A maximum of 8 instructions can be issued per cycle. Memory, address, and register file cross paths are used for communication between clusters. Most instructions introduce no delay slots, but multiply, load, and branch instructions introduce 1, 4 and 5 delay slots respectively. 6 general registers can be used as predication registers.

3.2 Compiler and Evaluation Methods

We use the TI C6x C compiler to compile the source code. Original procedures and integrated ones are compiled together with C6x compiler option '-o2 -mt'. The option '-o2' enables all optimizations except interprocedural ones. The option '-mt' helps software pipelining by performing aggressive memory anti-aliasing. It minimizes dependence bounds (i.e. RecMII), maximizing utilization for software pipelined loops. The C6x compiler has various features and is usually quite successful at producing efficient software pipelined code. For performance evaluation, we use the stand-alone simulator in Texas Instruments' Code Composer Studio (CCS) version 2.20. By enabling profiling, the execution cycles of original and integrated procedures are measured.

3.3 Procedure Selection and Integration

Four pairs of procedures are chosen from TI DSP/Image library [29] and MiBench benchmark suite [16] for integration. Among procedures which are relevant to main program workloads, various procedures with multiple loops (SWP or Non-SWP), conditionals or calls are selected to show the effectiveness of our methods. Table 1 summarizes selected procedures for experiments. For these four pairs of procedures, the iterative compilation method is applied and examined by generating all possible integrated procedures. Although analysis and estimation are automated, code transformation is performed manually for this work. The whole process can be automated by combining the compiler front- and back-ends with our analysis tools.

For the alignment set, we include only looping code as discussed in Section 2.3. For integration of looping code, we use loop jamming without any pretransformation to avoid complexity. General transformation techniques are performed; some further optimizations discussed in Section 2.5 are performed for simple loops with fixed iterations. For simulation, test inputs are generated, and variable-iteration loops are adjusted to execute at least 64 times. The profile used for static profitability estimation is assumed to be consistent with the actual run in order for the fair comparison of profitability measures from static estimation and simulation.

4. RESULTS AND ANALYSIS

Figure 5 shows the SCDGs of original procedure pairs used for experiments annotated with basic schedule and profile information. The profile information annotated on edges represents execution count or execution fraction. Below a code node, the schedule in-

Table 1: Summary of procedure pairs used for experiments

Case	Proc.A	Name	Proc.B	Name	Program	Benchmark suite
(a)	XF	sha_transform	BR	byte_reverse	sha	Mibench/security
(b)	DB	DSP_bexp	IH	IMG_histogram	N/A	TI DSP/Image Lib.
(c)	BCE	BF_cfb64_encrypt	BSK	BF_set_key	blowfish	Mibench/security
(d)	AC	Autocorrelation	CLTP	Calculation_of_the_LTP_parameters	gsm	Mibench/telecomm.

Table 2: SCDG characteristics of procedure pairs and STI results

Procedure Pairs	Proc.A			Proc.B			Integration Space			Total # of Compilations	Best Speedup
	L	P	R_A	L	P	R_B	$R_A \times R_B$	C_{AB}	L_{AB}		
(a) XF + BR	5	0	5	1	0	1	5	5	5	1	9.82%
(b) DB + IH	2	0	2	2	0	2	4	4	5	2	6.55%
(c) BCE + BSK	2	3	4	3	0	3	12	4	4	1	15.48%
(d) AC + CLTP	4	3	7	5	2	5	35	7	11	4	11.00%

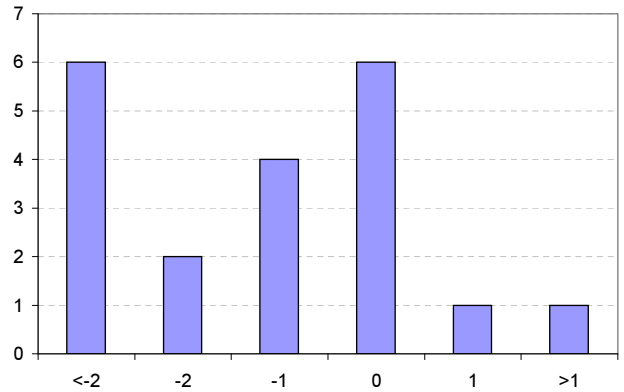
L: Loop nodes, P: Predicate nodes

formation is annotated with instruction and cycle count (*i/c*) and predicate and general registers used (*p-g*). The letter ‘S’ in a loop node indicates that it is software pipelined and ‘call’ appears in the annotation if there is a subroutine call inside a code node. Only code nodes under loop nodes are annotated because only combinations of those are considered for integration.

Table 2 summarizes the characteristics of original procedures, integration space and result of the iterative algorithm runs. The L, P and R_A (R_B) columns represent number of loop nodes, predicate nodes and code nodes located under loop nodes. The C_{AB} and L_{AB} columns show the number of combinations and alignments for integration. The actual combination set used for integration is a subset of the whole combination set (i.e. $C_{AB} \subset R_A \times R_B$) because some combinations such as SWP/Non-SWP pairs, ones with heavy register pressure are discarded by the estimation process. The last two columns show the number of compilations performed by the iterative algorithm and the speedup of the integrated procedure finally selected. While only 1 compilation is needed for case (a) and (c), 2 and 4 compilations are performed for case (b) and (d). Compared with the total number of all possible alignments (L_{AB} column), there are significant compilation savings by our method.

Figure 6 shows different profitability measures discussed in Section 2.6 obtained after generating and compiling all possible integrated procedures. For convenience, the AP is measured by simulation instead of static execution time measurement described in Section 2.2 because they are basically the same when the profile is consistent with the actual run. Though the EP is updated after every compilation cycle, this figure depicts the initial EP computed by the static estimation with no feedback. Alignments (i.e. integrated procedures) are numbered by the descending order of their initial EPs from the left. The alignment finally selected by the iterative algorithm (alignment 1, 2, 1 and 8 for each case) is marked with its % speedup.

The most important observation from this figure is that the procedure selected by our method is the actual best one for every case. The marked procedure shows the highest AP after actual compilation and simulation. In case (a) and (c), the alignment with the largest initial EP (i.e. alignment 1) is chosen as the best after compilation of itself. Recall that the algorithm decides whether it continues or not based on a comparison of EP and XP. In both cases, the iteration stops after the first compilation because $XP_1 > EP_2$. In case (b), one more compilation cycle is performed because $XP_1 < EP_2$ but $XP_2 > EP_3$. Regarding relationship between EP and XP, they show similar trends, meaning that static estimation works well for case (a), (b) and (c). The inequality $EP_i \geq XP_i$, the founda-


Figure 7: Histogram for schedule cycle count errors between pseudo-scheduling and real scheduling results

tion of decision mechanism, holds for all alignments in these three cases.

The case (d) is interesting because EP and XP show a significant difference for the first 5 alignments. This is due to unexpected errors from static estimation. Two combinations of SWP loops, which were estimated to be highly beneficial, turn out non-beneficial after actual compilation because SWP fails after integration due to high register pressure. For these loop combinations, the estimated number of used registers is slightly less than the number of physical registers. In such cases, SWP either succeeds or fails depending on the real schedule. A large estimation error is unavoidable when SWP fails because pseudo-scheduling works under the assumption that SWP still succeeds after integration.

When a large error occurs, the feedback mechanism plays a key role of removing non-beneficial combinations then re-ranking the alignments. When we see the details of compilation cycles for case (d), alignments 1 through 5 which have negative XP are removed after compilation of 1 and 3. The alignments 6 and 8 are then compiled based on updated EPs and 8 is chosen as the best after 4 total compilation cycles. This is clear advantage over 11 compilation cycles considering 11 possible alignments.

Figure 7 shows the histogram for schedule cycle count errors between pseudo-scheduling and real scheduling results of all 20 code combinations included in 4 cases. Most errors are negative indicating that pseudo-scheduling is more optimistic than real schedule. The errors are within 2 cycles for 12 combinations including 6 ex-

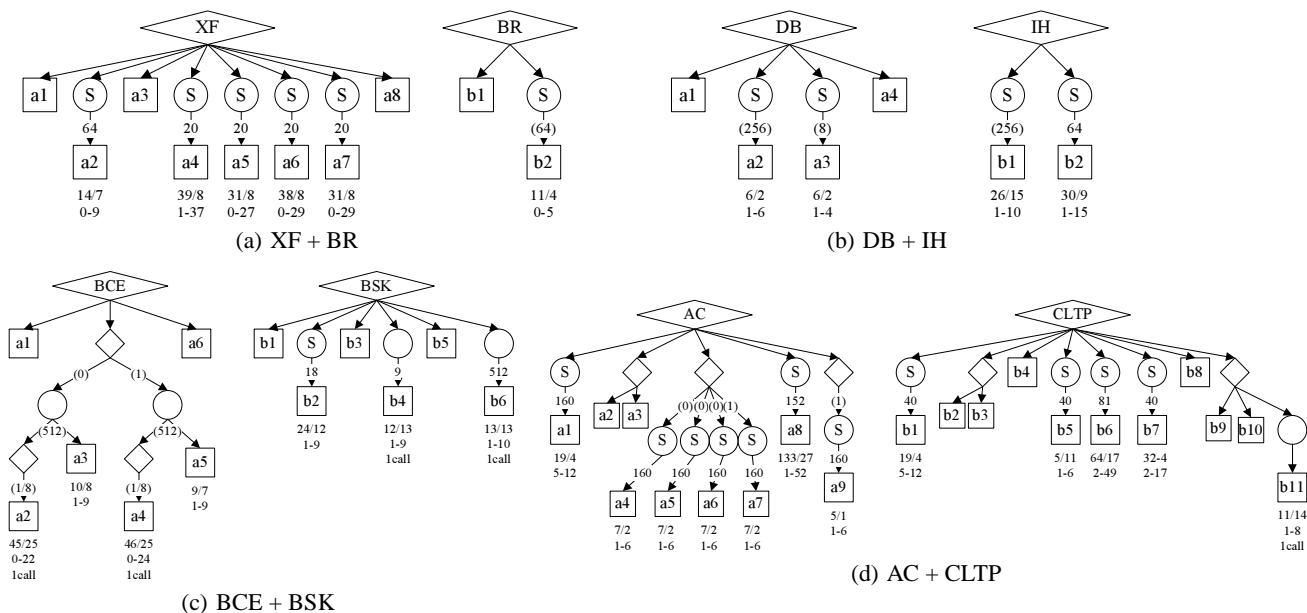


Figure 5: SCDGs of original procedure pairs with basic annotations

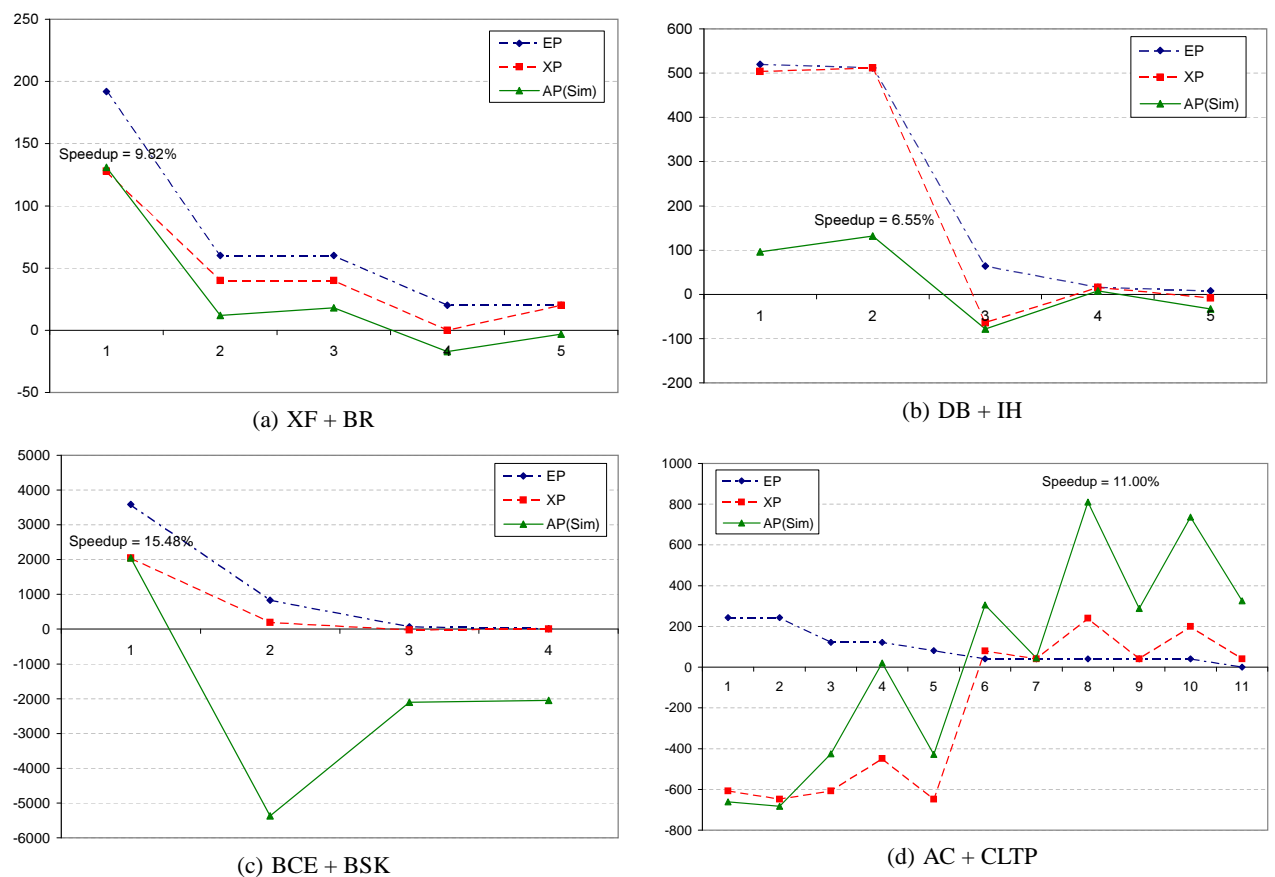


Figure 6: Different profitability measures of all alignments for each integration case

actly accurate ones. Among 6 combinations in which errors are less than -2, two correspond to the SWP loop combinations in case (d) where SWP fails after integration as discussed above. The other 4 combinations come from the case (c) where the integrated code includes subroutine calls. The errors are relatively large because pseudo-scheduling does not treat a call instruction specially while the real compiler does. However, the errors occur in all combinations so they do not affect the EP and XP relationship.

Experimental results demonstrate that the static profitability estimation via pseudo-scheduling works well despite some exceptions. The iterative compilation algorithm compensates for possible estimation inaccuracy and finds the best integration scenario with reasonable efficiency. When estimation works well, the algorithm effectively filters out the integrated procedures for which compilation is unnecessary by comparing realized profitability (XP) with potential profitability (EP). For the cases where estimation is not good enough, the feedback mechanism in the algorithm enables convergence of estimated profitability (EP) with the real profitability (XP) by applying previous compilation results to the future alignments.

5. RELATED WORK

5.1 Software Pipelining

Software Pipelining (SWP) [12, 25, 26] is a scheduling method to run different iterations of the same loop in parallel. However, software pipelining can suffer or fail when confronted with complex control flow, excessive register pressure, or tight loop-carried dependences. Much work has been performed to make software pipelining perform better on code with multiple control-flow paths.

Hierarchical reduction [21] merges conditional constructs into pseudo-operations and list schedules both conditional paths. If-conversion [3] converts control dependences into data dependences. Enhanced modulo scheduling [30, 31] begins with if-conversion to enable modulo scheduling then renames overlapping register lifetimes and finally performs reverse if-conversion.

STI improves the performance of SWP for control-intensive loops [28]. It increases the number of independent instructions visible to the compiler while effectively reducing loop overhead (through loop jamming). This enables the compiler to use existing SWP methods to create more efficient schedules.

5.2 Global Acyclic Scheduling

Compilers try to extract more ILP by forming bigger scheduling regions for acyclic code through merging multiple basic blocks. In trace scheduling [14], compilation proceeds by selecting a likely path of execution, called a trace. Superblock scheduling [11, 20] forms superblocks, regions with a single entrance and (possibly) multiple exits. A hyperblock [22] is a set of predicated basic blocks, in which control may only enter from the top, but may exit from one or more locations. Treeregion scheduling [18] forms a treeregion, single-entry/multiple-exit global scheduling region which consists of basic blocks with control flow forming a tree.

In region scheduling [15, 5], a program is divided into regions containing statements requiring the same control conditions via the Program Dependence Graph (PDG). Guided by the estimates of the parallelism present in the program regions, the region scheduler repeatedly transform the PDG, uncovering potential parallelism in regions.

STI assists the compilers to exploit parallelism across different procedures by merging them [28]. STI also allows arbitrary alignment of instructions or code by control flow transformations and code motion so that compilers use independent instructions efficiently.

5.3 Loop Transformations

Loop Jamming (or fusion) and unrolling are well-known optimizations for reducing loop overhead. Unroll-and-jam [2, 1, 7, 6] can increase the parallelism of an innermost loop in a loop nest. This is especially useful for software pipelining as it exposes more independent instructions, allowing creation of a more efficient schedule [9]. Unrolling factors have been determined analytically [8].

Loop fusion, unrolling, and unroll-and-jam have been used to distribute independent instructions across clusters in a VLIW architecture to minimize the impact of the inter-cluster communication delays [23, 24]. The technique called Deep Jam also uses loop jamming to increase ILP [10]. It recursively applies combinations of loop jamming and code duplication for inner loops and conditionals. However, transformation is limited to threads (threadlets) with the identical control flow while STI works both identical and heterogeneous control structures.

STI is different from these loop-oriented transformations in two ways. First, STI merges separate procedures, increasing the number of independent instructions within the compiler's scope. Second, STI distributes instructions or code regions to locations with idle resources, not just within loops. It does this with code motion as well as loop transformations (peeling, unrolling, splitting, and fusing).

5.4 Procedure Inlining

Procedure inlining (inline expansion) is a widely researched and accepted means to enable whole program optimization. It improves compiler analysis and enables other optimizations as well as instruction scheduling [4, 19, 13]. Way [33, 32] extends region-based compilation [17] to perform inlining and cloning based on demand and run-time profile information.

STI differs with procedure inlining techniques in two ways. First, overlap of the code of independent procedures by STI is not limited to the callees of the same procedure. Second, STI does not only merge two procedures but also transforms control structures so that the compiler generate an efficient schedule.

5.5 Software Thread Integration with Software Pipelining

A method for using STI to improve the performance of looping code on a VLIW DSP was proposed [28]. Efficient code transformation techniques using loop unrolling and loop jamming were defined depending on software pipelining (SWP) characteristics – SWP-Good, SWP-Poor and SWP-Fail – of loops to be integrated. It was shown that STI complements SWP by improving loops which benefit little from SWP.

Although the potential benefit of STI was shown, no method of high-level integration selection and guidance was provided, leaving unanswered the question of how to integrate two arbitrary procedures. This paper focuses on methods to find the best integration scenario from arbitrary code.

6. CONCLUSIONS

In this paper we propose an efficient method for finding the best integration scenario given two independent procedures with arbitrary control flow and utilization. Static profitability estimation relying on resource modeling, pseudo-scheduling and register pressure estimation allows various integration scenarios to be compared and ranked, identifying more beneficial ones. The iterative compilation algorithm compensates for possible estimation inaccuracy by decision and feedback mechanism, enabling quick selection of the best scenario with the least compilations. By experimental runs on

TI C64x DSP with code examples from TI DSP/Image library and MiBench, we demonstrate that our estimation shows reasonable accuracy for most cases with a few exceptions and the iterative algorithm finds the best integrated procedure with limited compilations despite difficulty of accurate estimation.

Future work includes incorporating dataflow analysis which improves estimation accuracy and identifies more legal alignments, and extending the method to more complicated cases where the code includes arbitrary number of procedures with multiple call hierarchies.

7. REFERENCES

- [1] A. Aiken and A. Nicolau. Loop quantization: an analysis and algorithm. Technical report, Cornell University, Ithaca, NY, USA, 1987.
- [2] F. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.
- [3] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.
- [4] R. Allen and S. Johnson. Compiling c for vectorization, parallelization, and inline expansion. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (PLDI '88)*, pages 241–249, New York, NY, USA, 1988. ACM Press.
- [5] V. H. Allen, J. Janardhan, R. M. Lee, and M. Srinivas. Enhanced region scheduling on a program dependence graph. In *Proceedings of the 25th annual international symposium on Microarchitecture (MICRO 25)*, pages 72–80, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
- [6] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI '90)*, pages 53–65, New York, NY, USA, 1990. ACM Press.
- [7] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 1988.
- [8] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proceedings of 29th Hawaii International Conference on System Sciences*, Jan. 1996.
- [9] S. Carr and Y. Guan. Unroll-and-jam using uniformly generated sets. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 349–357. IEEE Computer Society, 1997.
- [10] P. Carribault, A. Cohen, and W. Jalby. Deep jam: Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications. In *Proceedings of 14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005)*, pages 291 – 302, Sept. 2005.
- [11] P. P. Chang, N. J. Warter, S. Mahlke, W. Y. Chen, and W. W. Hwu. Three superblock scheduling models for superscalar and superpipelined processors. Technical report, University of Illinois, Urbana, IL, Dec. 1991.
- [12] A. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *IEEE Computer*, 14(3):18–27, 1981.
- [13] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–102, 1992.
- [14] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):278–490, 1981.
- [15] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, 1990.
- [16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001.
- [17] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: an introduction and motivation. In *Proceedings of the 28th annual international symposium on Microarchitecture (MICRO 28)*, pages 158–168. IEEE Computer Society Press, 1995.
- [18] W. Havanki, S. Banerjia, and T. Conte. Treeregion scheduling for wide issue processors. In *Proceedings of the The Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, page 266, Washington, DC, USA, 1998. IEEE Computer Society.
- [19] W. W. Hwu and P. P. Chang. Inline function expansion for compiling c programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation (PLDI '89)*, pages 246–257, New York, NY, USA, 1989. ACM Press.
- [20] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.
- [21] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (PLDI '88)*, pages 318–328. ACM Press, 1988.
- [22] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. *SIGMICRO Newsletter*, 23(1-2):45–54, 1992.
- [23] Y. Qian, S. Carr, and P. Sweany. Loop fusion for clustered VLIW architectures. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems (LCTES/SCOPE '02)*, pages 112–119. ACM Press, 2002.
- [24] Y. Qian, S. Carr, and P. H. Sweany. Optimizing loop performance for clustered VLIW architectures. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 271–280. IEEE Computer Society, 2002.
- [25] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th annual workshop on Microprogramming (MICRO 14)*, pages 183–198, Piscataway, NJ, USA, 1981. IEEE Press.
- [26] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *Proceedings of the 9th annual symposium on Computer Architecture (ISCA '82)*, pages

- 131–139, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.
- [27] W. So and A. G. Dean. Procedure cloning and integration for converting parallelism from coarse to fine grain. In *Proceedings of Seventh Workshop on Interaction between Compilers and Computer Architecture (INTERACT-7)*, pages 27–36. IEEE Computer Society, Feb. 2003.
- [28] W. So and A. G. Dean. Complementing software pipelining with software thread integration. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '05)*. ACM Press, 2005.
- [29] Texas Instruments. *TMS320C64x DSP Library Programmer's Reference*, Apr. 2002.
- [30] N. J. Warter, J. W. Bockhaus, G. E. Haab, and K. Subramanian. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Portland, Oregon, 1992. ACM and IEEE.
- [31] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation (PLDI '93)*, pages 290–299, New York, NY, USA, 1993. ACM Press.
- [32] T. Way, B. Breech, and L. Pollock. Demand-driven inlining heuristics in a region-based optimizing compiler for ILP architectures. In *Proceedings of the 2001 IASTED International Conference on Parallel and Distributed Computing and systems (PDCS '01)*, pages 90–95, Anaheim, CA, USA, Aug. 2001.
- [33] T. Way and L. Pollock. Using path spectra to direct function cloning. In *Workshop on Profile and Feedback-Directed Compilation*, pages 40–47, Oct. 1998.