IISc-CSA-2002-4

Computer Science and Automation Department
Indian Institute of Science
http://144.16.67.13/~techrep/index.html

A Region Based Framework for Combined Register Allocation and Instruction Scheduling for EPIC Architectures

Pradeep Jain, Yogesh Jain and K. Gopinath
Department of Computer Science and Automation
Indian Institute of Science, Bangalore
gopi@csa.iisc.ernet.in

Abstract

The Trimaran compiler infrastructure has been developed for supporting state of art research in compiling Instruction Level Parallelism (ILP) architectures, especially those based on Explicitly Parallel Instruction Computing (EPIC) and for research in instruction scheduling, register allocation and machine-dependent optimizations.

The current framework in Trimaran is not suitable for experimenting with techniques in which the instruction scheduling and register allocation interact. This is mainly due to the horizontal model of compilation employed in these optimizations where one optimization phase is carried out on all the regions of the function before proceeding to the next phase. The vertical model of compilation, in which several phases can be carried out simultaneously on a region before proceeding to the next region, is needed for exploiting more focussed optimizations. We have redesigned the framework for the ordering of execution of scheduling and register allocation phases in Trimaran to meet the above goals.

Using the framework, we have combined instruction scheduling and register allocation by first using a profile-insensitive list scheduler, then incorporating register pressure to make it sensitive to register allocation costs and finally making it profile-sensitive.

We give the design of the framework and the performance results of our implementation.

1 Introduction

It is well recognized that the separation between the register allocation and instruction scheduling phases leads to significant problems[14] [13] [15], such as poor optimizations for cases that are ill-suited to the specific phase-ordering selected by the compiler, and additional conceptual and software-engineering complexities in trying to adjust one phase to take into account cost considerations from the other. If instruction scheduling is performed first it may produce a schedule that will require values in registers to be spilled. The instructions for performing the spilling must be inserted into the schedule, usually requiring a second scheduling phase. If register allocation is performed first it can reduce the amount of parallelism available for scheduling due to the introduction of false dependencies on registers. Recent work has developed methods to incorporate some information from one phase into the other[16].

A specification for a parametrizable machine architecture, HPL-PD [2], has been developed at HP Labs to further research into VLIW and superscalar optimizations. Also, the Trimaran compiler infrastructure [1] is being developed for supporting state of art research in compiling for ILP architectures. The system is currently oriented towards Explicitly Parallel Instruction computing(EPIC) architectures, and supports compiler research in what is typically considered to be "back-end" techniques such as instruction scheduling, register allocation and machine-dependent optimizations.

The current framework in Trimaran in not suitable for experimenting with techniques in which the instruction scheduling and register allocations interact. This is mainly due to the horizontal model of compilation in these phases where a phase is carried out on all the region of the function before proceeding to the next phase. The vertical model of compilation, in which several phases can be carried out simultaneously on a region before proceeding to the next region, is needed for better compilation. In this paper, we discuss our redesign of the framework through reordering the execution of scheduling and register allocation phases in Trimaran to enable a vertical model. We study the compilation and execution times of the framework and interaction with other aspects like modulo and meld scheduling.

1.1 Organization of the Paper

The region-based approach is compared vis-a-vis the function based approach in section 2. Section 3 gives an overview of Trimaran compiler infrastructure. Meld Scheduling and Region based register allocation are explained in Section 4. Section 5 gives the details of our combined region based instruction scheduling and register allocation and the issues involved in its implementation. The performance evaluation of our implementation is given in Section 6. Section 7 concludes with scope for future work.

2 Compilation Models

We briefly compare the traditional function based approach (also called horizontal model of compilation) with region based compilation approach (also called vertical model of compilation). For details, see[11] [12]. Traditionally, the compilation process has been built assuming functions as a unit of compilation. Figure 1 illustrates the process of compiling a program of n functions. However it suffers from the following limitations: the scope of compilation/optimization is limited to a procedure and not across critical paths spanning procedures; lack of control of compilation unit size; inability to reuse analysis information and restricted use of profile information.

2.1 Region based compilation

The drawbacks of function based compilation can be addressed by allowing the compiler to repartition the program into a new set of compilation units, called regions. A region is defined as an arbitrary collection of basic blocks selected to be compiled as a unit. Each region may be compiled completely before compilation proceeds to the next region. In this sense, the fundamental mode of compilation has not been altered and all previously proposed function-oriented compiler transformations may be applied. The benefits are the use of dynamic behavior of a program and control over the size of compilation unit.

2.1.1 Issues in region based compilation

Region selection A region-based compiler begins by repartitioning the program into regions. The region selector may select one region at a time or it may select all regions a priori, before the compilation process begins on any region. Following criteria can be used for the region selection :

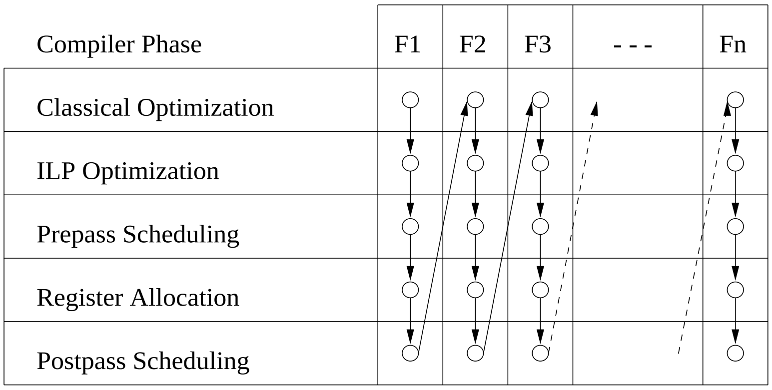

Figure 1: Block diagram of function-based compilation

- Execution frequency information of operations
- Memory access information
- Structure of region e.g. basicblock, superblock[8], hyperblock[9], loops etc.
- Maximum size of the region (to limit the complexity of region processing)

Region selection can be performed on a single procedure or it can span more procedures. The scope of region selection can also be increased beyond a procedure by performing function-inlining before region selection.

Phase ordering In region based compilation, the phases of compilation can be carried out on the regions in any order. Only the constraints arising due to the properties of the phases themselves should be obeyed. Different regions can be in two completely different phases of the compilation process at the same time. This approach is very flexible in comparison to the function based approach in which each phase of compilation is applied to every basic block in the function before the next phase begins.

Incremental analysis Each compiler phase performs certain global analysis on all the regions before starting the processing of individual regions. However, as the compilation proceeds, this information might be corrupted in certain cases and would require to be updated. This requires some form of incremental analysis.

Region boundary conditions Separate compilation of a program using a traditional function-based compiler is facilitated by the fact that the boundary conditions of a function are fixed. The variables live across the single entry point and single exit point of a function are well defined by the parameter passing convention. However, a region is an arbitrary partition of the program's control flow graph. The liveness and other necessary information is required at the boundary of these regions. This information can also change dynamically as the compilation proceeds.

3 Trimaran Compiler Infrastructure

The Trimaran compiler infrastructure is comprised of the following components (for details see appendix):

- A machine description facility, MDES, for describing ILP architectures [4].
- A parameterized ILP Architecture called HPL-PD [2].
- A compiler front-end, called IMPACT, for C. It performs parsing, type checking, and a large suite of high-level (i.e. machine independent) optimizations
- A compiler back-end, called Elcor, parameterized by a machine description, performing instruction scheduling, register allocation, and machine-dependent optimizations. Each stage of the back-end may easily be replaced or modified by a compiler researcher [5].
- An extensible IR (intermediate representation) which has both an internal and textual representation, with conversion routines between the two. The textual language is called Rebel. This IR supports modern compiler techniques by representing control flow, data and control dependence, and many other attributes. It is easy to use in its internal representation and its textual representation.

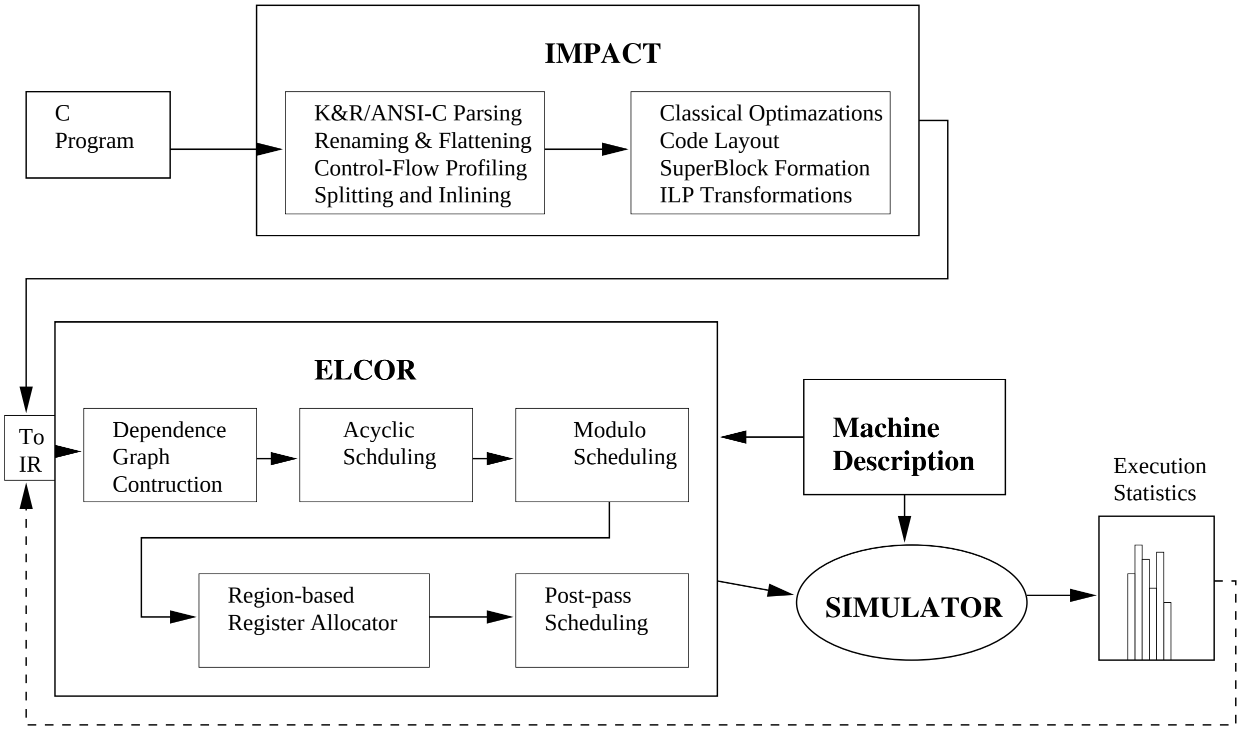

Figure 2: Trimaran Compiler Infrastructure

- A cycle-level simulator of the HPL-PD architecture which is configurable by a machine description and provides run-time information on execution time, branch frequencies, and resource utilization. This information can be used for profile-driven optimizations as well as to provide validation of new optimizations.
- An integrated Graphical User Interface (GUI) for configuring and running the Trimaran system. Included in the GUI are tools for the graphical visualization of the program IR and of the performance results.

For the purposes of this paper, the following infrastructure is relevant:

Modulo Scheduling of Counted Loops Loop scheduler consists of two parts. Modulo scheduler allocates resources for the loop kernel subject to an initiation interval. Stage scheduler moves operations across stages in order to reduce register usage of the loop. The existing implementation of loop scheduling generates kernel only code, which requires support for predication and rotating register files.

Acyclic Scheduling of Superblocks/Hyperblocks There are three variations of acyclic scheduling in Elcor viz. cycle-scheduler, backtracking-scheduler and meld-scheduler. Cycle scheduler generates a schedule by constructing instructions from operations for each issue cycle in order. Backtracking scheduler is a modified version of the cycle scheduler which can either do limited backtracking to only support scheduling of branch operations with branch delay slots, or do unlimited backtracking. Meld scheduler is a modified version of the cycle scheduler that can propagate operation latency constraints across scheduling region boundaries. This results in tighter scheduling of operations across region boundaries.

Rotating Register Allocator When a counted loop is software pipelined, a set of virtual registers in the loop are designated as rotating registers. Rotating register allocator allocates such registers to the rotating register files right after modulo scheduling. Stage scheduling can be used after modulo scheduling to reduce the rotating register requirements of a loop. The remaining registers are allocated to the static register file after scalar scheduling of the rest of the program.

Region Based Scalar Register Allocator This allocator is based on Chow and Hennessy's approach[19]. The cost model is based on caller-callee register information.

4 Meld Scheduling & Region Based Register Allocation

The following is the outline of the current implementation of instruction scheduling and register allocation in Trimaran.

```
horizontal_sched_RA(Procedure P) {
    pre_pass_schedule_all_regions(P)

    register_allocate_all_regions(P)
    generate_reconcile_code(P)

    post_pass_schedule_all_regions(P)
}

pre_pass_schedule_all_regions(Procedure P) {
    for each region R in P
        pre_pass_schedule(R)
}

register_allocate_all_regions(Procedure P) {
    for each region R in P
        register_allocate(R)
}

post_pass_schedule_all_regions(Procedure P) {
    for each region R in P
        post_pass_schedule(R)
}
```

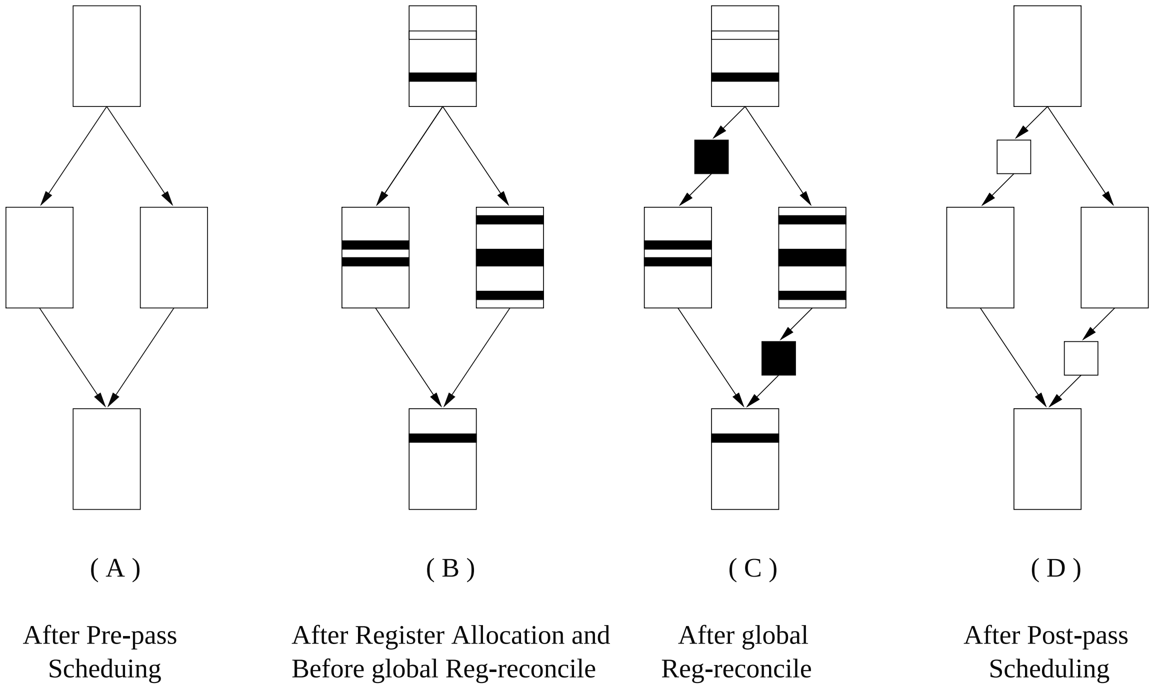

Figure 3: Horizontal compilation: IR graph at various stages

Figure 3 shows the state of the graph at various stages of the compilation. The rectangular blocks represent the original regions of the procedure. The small squares represent the additional basicblocks added to accommodate the patch-up code. Black regions indicate the unscheduled portion. (B) shows the code added due to intra-region register reconcile. (C) shows the patch-up blocks added due to global register reconciliation.

The pre-pass scheduling phase is function based and is performed on all the regions of the function at a time. This phase considers the operands to be assigned to virtual registers and not physical registers. Register allocation follows after pre-pass scheduling phase. Register allocator itself is region based, but it cannot be mixed with the scheduling phase because of the function based nature of scheduling. Spill code is generated within the regions during this phase. Also, new basic blocks are added to the graph to accommodate patch-up code for inter-region live operands.

The graph seen by the post-pass scheduling phase is different from the one seen by pre-pass scheduling phase as it contains additional spill code operations within the regions and it contains extra basic blocks which contain patch-up operations for inter-region live operands. Post-pass scheduling is similar to pre-pass scheduling in all other respects.

The following sections give a brief introduction to meld-scheduling and region based register allocation. This will help in understanding the implementation of our region based approach for combined instruction scheduling and register allocation.

4.1 Meld Scheduling

Scheduling algorithms are usually oblivious of constraints coming in from blocks or regions that are already scheduled and hence may make scheduling decisions that lead to poor schedules. It is however possible to improve the schedules if the constraints from already scheduled regions are propagated to the regions yet to be scheduled. Meld Scheduling is an example of this approach[17].

In a non-interlocked machine, the processor does not interlock to ensure that the inputs are available before issuing an operation. For such machines, the compiler schedules code to guarantee that an operation completes before a dependent operation issues. Within a scheduling region, the scheduler delays the issue of a dependent operation to ensure that its inputs are available. Across scheduling regions, a scheduler must ensure that certain constraints are satisfied on entry or exit of a region. For instance, one convention is to assume that on entry all operations have their inputs available. In this case, the scheduler must guarantee that all operations complete before control is transferred to another region. In contrast, a meld scheduler generates latency constraints imposed by scheduled regions, propagates constraints to the boundaries of a region to be scheduled, and translates these constraints to edge constraints recognized by the local region scheduler. We omit further discussion as additional details are available in [17].

4.2 Region Based Register Allocator

Trimaran has a region based register-allocator[19] with the following features: frequency-based priority coloring (in the spirit of Chow and Hennessy[18]), region-based with region reconciliation, fine grained live ranges with predicated instructions and live-ranges. In the following sections, we give only the details which are relevant to understand our algorithm and avoid other details.

4.2.1 Live-range construction

The live-range is the basic unit for register assignment. It is an isolated and contiguous group of nodes in the control flow graph in which a program data value is live and repeatedly accessed. In our case, the nodes are the operations in the program. A live-range typically begins at definition points and terminates at last uses. Each live-range is assigned to a single register. In the course of allocation, a live-range may be split into two or more smaller live-ranges. Subsequently, each of the smaller live-ranges can be assigned to a different register.

Since the analysis is for predicated code, the live-range is a list of $<$ Operation, Pred_cookie $>$, where the Operation is the the operation in which the operand is referenced and Pred_cookie is the predicate expression associated with operation. The construction of live-range is similar to other data-flow analysis like liveness and reaching definition analysis within a region. However, here the analysis also takes predicates of operations into consideration. The live-range also stores additional information i.e interfering live-ranges, forbidden registers, reference operands of definitions and uses, priority information. This information is required for the register allocation.

4.2.2 Classification of live-ranges

- *A* live-ranges: These are strictly local to the region. These are register allocated by intra-region register allocator and do not require global reconciliation.

- *Bin/Bout* live-ranges: These have VRs that are live-in/live-out from/to other regions (and may be at the same time both live-in/live-out but they cannot be pass-through). These are register allocated by intra-region register allocator, but their bindings must be propagated for global reconciliation
- *C* live-ranges: These are the transparent ones. They are resolved on the basis of the binding of other adjacent global live-ranges

All A and B type live-ranges are resolved (bound or spilled) during intra-region register allocation. All C type live-ranges are delayed.

4.2.3 Intra-region register allocation

```
driver(Region R) {
    for each subregion SR in R by frequency order
        if SR is base intra-region
            intra_region_register_allocation(SR)
        else driver(SR)

        resolve_inter_region_live_ranges(SR)
        mark_callee_used_registers(SR)

    reconcile_inter_region_live_ranges(R)
    reconcile_callee_used_registers(R)
}

intra_region_register_allocation(Region R) {
    live-range-set LRS = construct-live-ranges(R)

    for each filetype F
        live-range-set LRS_F = live ranges of filetype F in LRS

        set-interference-graph(LRS_F);
        set-priority(LRS_F);

        for each live-range LR in LRS_F by priority order
            if LR is pass-through, delay it
            else if suitable reg REG is found, bind LR to REG
            else if spilling is suitable, spill LR
            else split LR
}
```

There are separate register-files corresponding to different file-types. So, the live-ranges belonging to different file-types cannot interfere. In order to reduce the complexity, we process the live-ranges belonging to a single file-type at a time.

First the interference graph is formed and the priorities are assigned to the live-ranges. The unconstrained live-ranges are separated and they are processed only after all constrained live-ranges. The constrained live-ranges are visited in the priority order. Live-ranges are bound or spilled depending on the suitable cost function and the availability of registers.

4.2.4 Inter-region register allocation

After all the A and B type live-ranges are resolved, we resolve the inter-region live-ranges as shown below:

```
resolve_inter_region_live_ranges (Region R) {
    for each non-intra live-range LR of R
        add LR to global_LR

        if LR is pass-through live-range
            if all adjacent live-ranges of LR in global_LR are delayed
                delay LR
            else
                resolve LR using most suitable adjacent live-range
                adj_LR of LR in global_LR

        propagate_bindings(LR)
```

```
}

propagate_bindings(LiveRange LR) {
    if LR is not pass-through live-range
        for all adjacent live-ranges adj_LR of LR in global_LR
            Edge E = edge joining LR and adj_LR

            if source operation of E in is LR
                SRC = LR
                DEST = adj_LR
            else
                SRC = adj_LR
                DEST = LR

            if  SRC is spilled and DEST is spilled
                no patch-up code
            else if SRC is spilled and DEST is bound
                add code for load on E
            else if SRC is bound and DEST is spilled
                add code for save on E
            else if SRC is bound and DEST is bound
                add code for move on E

            add E to unordered_ops
}
```

global_LR is a set of all inter-region live-ranges and is used to find the adjacent global live-ranges. The bindings of B type live-ranges are propagated to the other adjacent global live-ranges. Since the C type live-ranges are not resolved in intra-region register allocation, we first try to resolve them. A C type live-range is resolved if there is at least one adjacent global live-range which is already resolved, otherwise it is marked as delayed.

The propagation of binding consists of adding the requisite patch-up information between the current live-range and all its global adjacent live-ranges. The patch-up information is added on the edge joining the two live-ranges. *unordered_ops* is the list of all the edges on which the patch-up information is stored.

After all the regions are processed, the patch-up code is generated using the patch-up information stored on the edges earlier.

```
reconcile_inter_region_live_ranges (Region R) {
    for each Edge E in unordered_ops
        if source operation of E is dummy branch
            add patch_up code in the region containing source
            operation of E

        else if destination operation of E has only one incoming edge
            add patch_up code in the region containing destination
            operation of E

        else
            create a new basicblock corresponding to E and add the patch_up
            code in this new block
}
```

4.2.5 Reconciliation of Callee-saved Registers

If the callee-saved registers are used by a procedure, it must save and store these registers at its entry and exit respectively. *mark_callee_used_registers()* marks all the callee-saved registers used by the register allocation. The reconciliation code to these registers is added to the epilogue and prologue regions of the procedure in *reconcile_callee_used_registers()*.

5 Combined Region Based Framework

We present below a region based framework for combining instruction scheduling and register allocation. Design decisions are presented below by using a step-wise refinement approach.

5.1 Stepwise Development of Framework

5.1.1 Horizontal Compilation Model

The current implementation of both register allocation and scheduling in Trimaran is based on horizontal compilation model as discussed in Section 4 (see also Figure 4). In this model, the compilation phases are completely decoupled. This has the following drawbacks:

- If the register allocator wants to use any information from the pre-pass scheduler, such information should be stored for all the regions. This increases memory requirements. Similarly, if the post-pass scheduler wants to use any information from the register allocator.
- Most of the flow-analysis and liveness analysis is common for pre-pass and post-pass scheduling phase. To reuse this information, it should be stored for all the regions. This again increases memory requirements. Currently it is being recalculated for post-pass scheduler phase. This increases the compilation time. This difference is considerable, because most of the analysis is complicated, especially in context of predicated code.

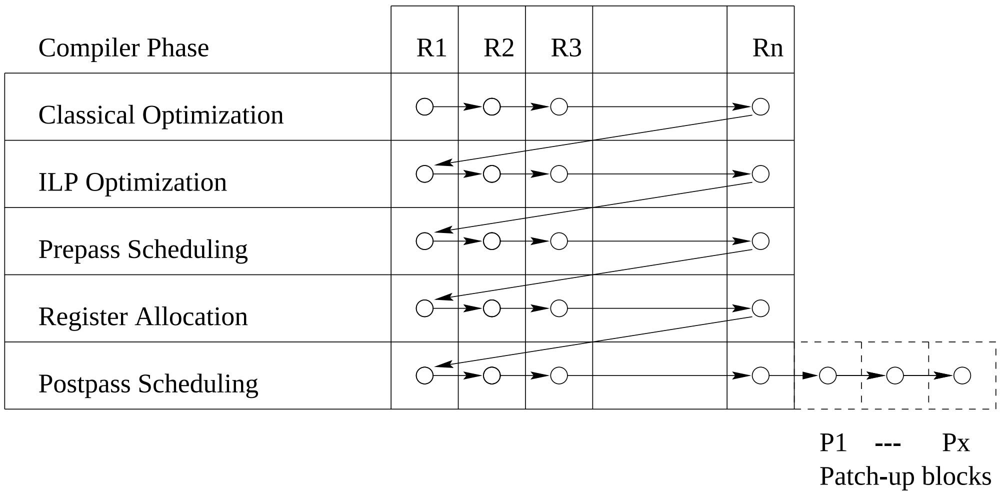

Figure 4: Block diagram for horizontal compilation model

5.1.2 Vertical Compilation Model I

In order to avoid storing the information for all the regions between the successive use of information, we call all the three phases on a region simultaneously. This reduces memory requirements. The Figure 5 shows the order of execution of compilation phases in this model. The following is the outline of this model:

```
vertical_sched_RA_model_1(Procedure P) {
    for each region R in Procedure P

        pre_pass_schedule(R)
        register_allocate(R)
        post_pass_schedule(R)

    generate_reconcile_code(P)
    schedule_patch_up_blocks(P)
}
```

During the register allocation phase, the actual reconcile code is not generated, only the information for reconcile code is stored on the corresponding edges. After all the original regions are processed, the reconcile code for inter-region live-ranges is generated. The new patchup basic blocks generated are then scheduled separately. In this model, the original regions are not scheduled after adding the patch-up blocks. This could result in an inferior schedule if meld-scheduling is enabled.

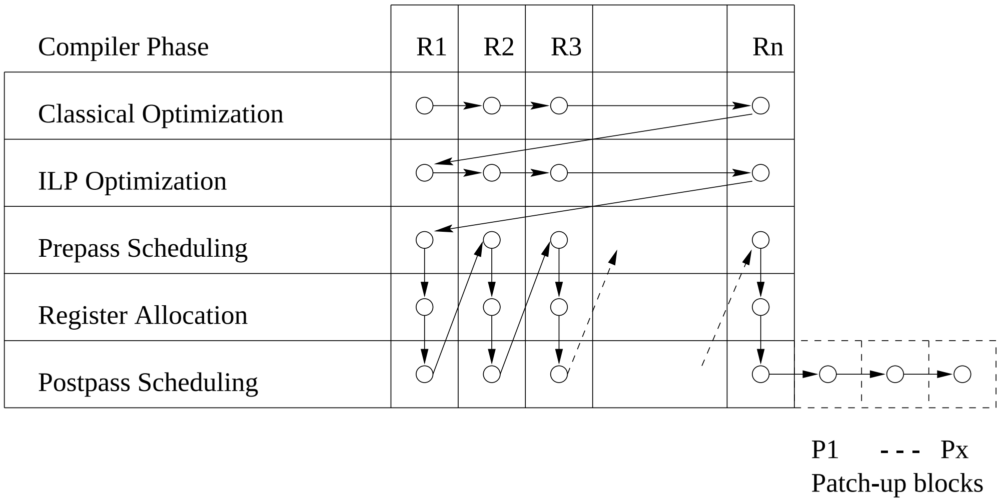

Figure 5: Block diagram for vertical compilation model I

5.1.3 Vertical Compilation Model II

In order to ensure that the presence of patch-up blocks is reflected in the scheduling of the original regions, an additional post-pass scheduling phase is introduced. The outline of the model is given below.

```
vertical_sched_RA_model_2(Procedure P) {
    for each region R in Procedure P
        pre_pass_schedule(R)
        register_allocate(R)
        post_pass_schedule(R)

    generate_reconcile_code(P)

    for each region R in Procedure P
        post_pass_schedule(R)
}
```

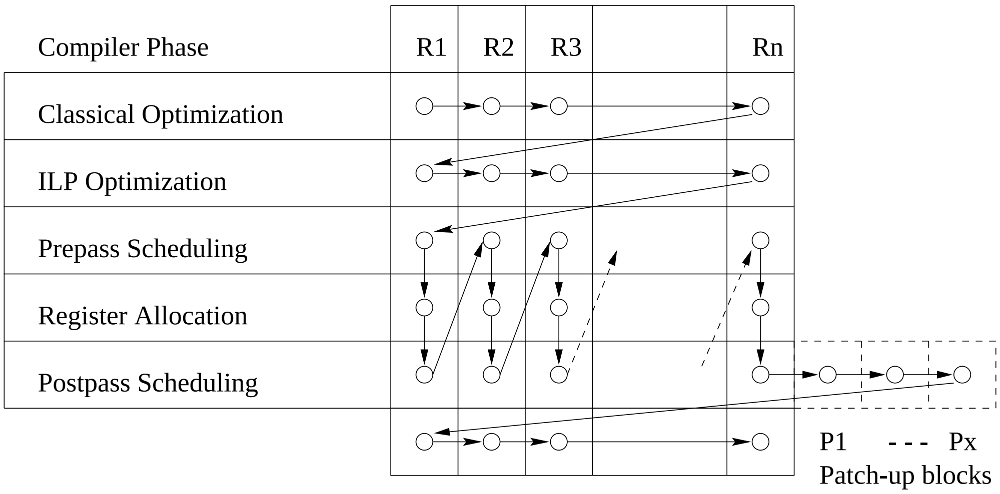

Figure 6: Block diagram for vertical compilation model II

The final schedule of this model will be as good as the horizontal model. But the scheduling of each region is done three times in this model in contrast to two times in the previous model. This is obviously costly in terms of compile time.

5.1.4 Vertical Compilation Model III

In the final model, reconcile code corresponding to a region is created immediately after its register allocation. The outline of this model are given below.

```
vertical_sched_RA_model_3(Procedure P) {
    for each region R in Procedure P
        pre_pass_schedule(R)

        register_allocate(R)
        generate_inter_region_reconcile_code(R)

        post_pass_schedule(R)
}
```

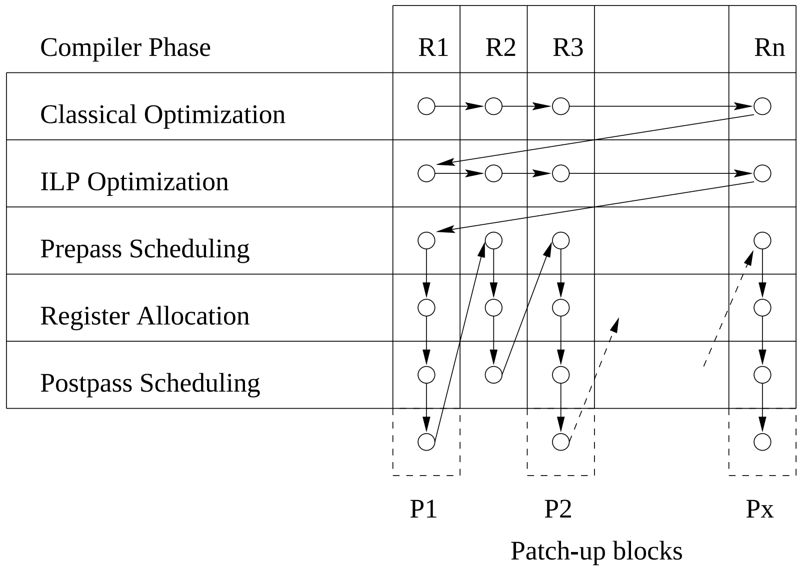

Figure 7: Block diagram for vertical compilation model III

5.2 Problems in vertical compilation model III and Solutions

5.2.1 Eager propagation of inter-region reconcile information

The register reconcile information is generated immediately after the register allocation of the region. It is desired that this information be propagated completely so that further processing can make use of it.

The post-pass scheduling of the region takes care of the intra-region patch-up code added during the register allocation of the region. Similarly, under certain conditions, the inter-region reconciliation code may be added to the region being processed itself. This is possible when the exit-op of the region is a dummy-branch or when there is only one incoming edge to the region. Post-pass scheduling takes care of such reconciliation code also.

In all other cases, extra patch-up blocks are added to accommodate the reconciliation code. These patch-up blocks are made similar to the original blocks in all respects, so that the meld-scheduling of the remaining regions takes into account the latency produced by these regions. Thus the meld-scheduling works with the most accurate latency information.

5.2.2 Handling of patchup basic blocks

When should we schedule the patchup basic blocks? For calculating the meld-constraints, the paths consisting of only scheduled blocks are considered. As shown in Figure 8, the meld-constraints from region A are propagated to region B if patch-up block P is scheduled. Otherwise, it is assumed that the scheduling of P will absorb the latency constraints of both regions A and B.

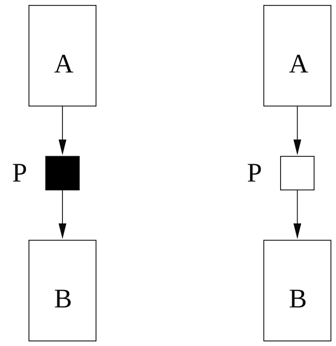

Figure 8: Meld-constraint propagation through patchup basic block

In the absence of P, the meld-constraints of region A are propagated to region B. This is not possible if P is present and is unscheduled. In short, the meld-constraints become more restricted if the patch-up blocks are left unscheduled. So the patchup basic blocks should be scheduled as soon as they are created.

5.2.3 Handling the pass-through live-ranges

Consider Figure 9. The dark line represents a global live-range which consists of local live-ranges *L1, L2, L3* and *L4*. Assume that *L2* and *L3* are pass-through live-ranges and the regions are register-allocated in the order *B - C - A - D*.

In the original implementation of register-allocation in Trimaran, the pass-through live-range is left unresolved until at least one of its adjacent global live-ranges is resolved. So, when B is register-allocated, *L2* is left unresolved. Similarly, when C is register-allocated, *L3* is left unresolved. When A is register-allocated, this binding is propagated to *L2* and then to *L3*. This propagation only adds the corresponding patch-up code on the edge and does not generate any patch-up block in the graph until the end of register allocation of all the regions.

In our implementation, the new patch-up block P is added corresponding to edge *e1* when C is processed. When A is register-allocated, the propagation of binding of *L1* might result in additional patchup code in P, This would require rescheduling of P. This cannot be allowed because earlier regions have used the meld-constraints based on the old schedule of P.

We avoid the above problem by not allowing the binding of the pass-through live-ranges to be delayed. For a pass-through live-range, if already resolved global adjacent live-range is not found, we spill it. This could result in spilling of some pass-through live-ranges which are successfully bound in original implementation in Trimaran. However, the performance is not affected much, as shown by our experimental results.

5.2.4 Hiding patchup basic blocks from register allocator with delayed passthroughs

As we pointed out earlier, the meld scheduling considers the patchup basic blocks to be similar to the original blocks. But how should the register allocator view the patch-up basic-blocks? When the resolution of pass-through live-ranges is delayed, as in the original implementation in Trimaran as pointed out in the previous section, there is a possibility of patch-up code being added to an edge in more than one visit.

Again consider Figure 9. If P is considered to be similar to the original blocks by the register allocator, then the patch-up block for the pass-through live-range is required on the edge *e3* and *e4*. Since P is already scheduled, new basicblocks would be required corresponding to *e3* and *e4*. This is obviously not a practical solution.

So for the purpose of register allocator, the patch-up blocks should be dealt differently. We follow a very simple approach. We assume that the register allocator views only the original edges of the graph and the patch-up blocks are added only corresponding to the original edges. However, the original edges must be deleted otherwise the meld-scheduling will bypass the patch-up block. So, we store the original edges of the graph in the following data structures before beginning to process the regions:

```
Map<Op*, List<Edge*> > original_inedges;
Map<Op*, List<Edge*> > original_outedges;
```

The register allocator reads the edges for the operations from the above data-structures instead of the actual edges of the operations.

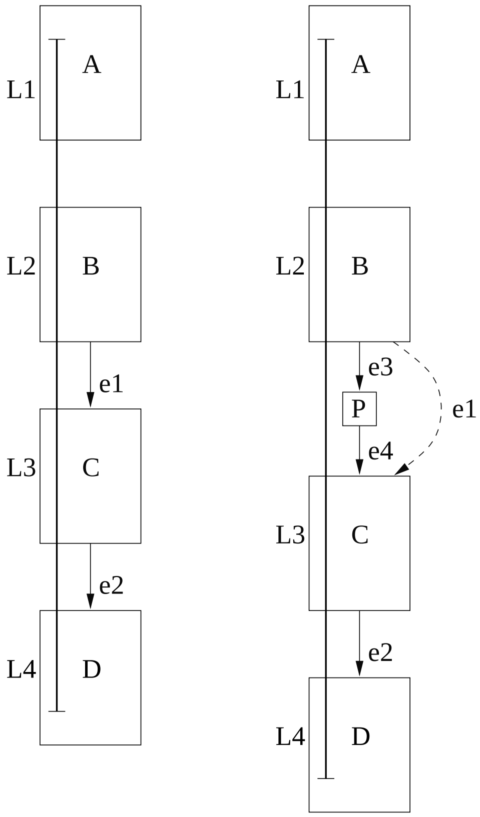

Figure 9: Reconciliation of pass-through live-ranges

The above approach is not required in our implementation. However, the design allows the study of handling of pass-through live-ranges based on the original implementation as explained in previous section.

5.2.5 Prologue and epilogue regions

If the callee-saved registers are used by the register allocator, reconcile code must be generated for them. It consists of adding the load and store operation in the prologue region and epilogue region of the procedure respectively. Since, the callee-saved registers used are known only after the register allocation of all the regions, the reconcile code is added to the epilogue and prologue region after processing all the region. As a result, these prologue and epilogue regions should be scheduled at the end. The same care is not required for these regions in the horizontal model because the reconcile code for the callee-saved registers is added in the global-register allocation phase and the post-pass scheduling of the prologue and the epilogue takes place later.

5.3 Final design

All the above issues are incorporated in our framework for combined region based instruction scheduling and register allocation. The outline of this model is given below.

```
vertical_sched_RA_model_final(Procedure P) {
    form a sorted list of original regions of the procedure P,
    called initial_regions

    for each region R in initial_regions
        for each entry_op OP of R
            bind < OP, list of in_edges of OP > to initial_inedges

        for each exit_op OP of R
            bind < OP, list of out_edges of OP > to initial_outedges

    for each region R in initial_regions
        pre_pass_schedule(R)

        register_allocate(R)
        generate_inter_region_reconcile_code(R)

        post_pass_schedule(R)

        generate_spill_code_for_used_callee_registers()
        schedule_prolog_and_epilog(R)
}
```

6 Combined Instruction Scheduling And Register Allocation

Using the framework, we have combined instruction scheduling and register allocation by first using a profile-insensitive list scheduler, then incorporating register pressure to make it sensitive to register allocation costs and finally making it profile-sensitive.

The outline of the approach is as follows:

- Use a profile-insensitive list scheduler, such as a critical path scheduler. This is equivalent to the list scheduling with the distance from the last exit as the list priority. This gives us a scheduling rank.
- Compute a register rank. We use the CRISP model from [16].
- Combine the two ranks to get a schedule sensitive to register pressure.
- Convert this profile-insensitive schedule to a profile-sensitive schedule that is also sensitive to register pressure. We use Chekuri's heuristic [23].

6.1 Register Rank

The CRISP register rank γ_R is defined as follows [16]: For each instruction v, define

$$Thick_1(v) \quad = \quad \sharp \text{ value ranges from an unrelated}$$

$$Thick_2(v) = \begin{array}{l} \text{instruction to a descendant of } v \\ \sharp \text{ value ranges from an ancestor of} \\ v \text{ to an unrelated instruction} \end{array}$$

The instructions are then sorted in increasing order of $\Delta_v = (Thick_1(v) - Thick_2(v))$. The register rank $\gamma_R(v)$ is the rank of instruction v in this sorted order. As is usual in ranking, instructions with the same Δ value are assigned the same rank. The crux of this definition is its ability to prioritize a node v relative to nodes that are incomparable to v, i.e., those that are neither an ancestor nor a descendant of v.

To understand the meaning of Δ_v, we can see that when $Thick_1(v)$ is large there is a large number of nodes independent of node v and it would be beneficial from register considerations to schedule node v later so as to shorten the live ranges of these nodes. Thus, we make a larger $Thick_1(v)$ value lead to a larger Δ_v rank . Conversely, if $Thick_2(v)$ is large, it would be beneficial to schedule v earlier. Thus, we make a larger $Thick_2(v)$ value lead to a smaller Δ_v rank.

Scheduling rank γ_S : We can use the rank function from any suitable instruction scheduling algorithm as the scheduling rank. In our implementation, we used the critical path rank function as the scheduling rank, γ_S.

For any choice of parameters $\alpha, \beta \in [0, 1]$ such that $\alpha + \beta = 1$, we now have the following heuristic.

(α, β)-**Combined Algorithm :** This heuristic creates a combined rank function $\gamma = \alpha\gamma_S + \beta\gamma_R$, and orders the instructions into a list in increasing order of rank. It then runs the greedy list scheduling algorithm using this list to obtain a schedule.

The heuristic described above is for a fixed α or β value. The parameter α (or β) can be viewed as a tuning parameter, and can be set differently for different processor architectures, size of basicblocks, superblocks, and hyperblocks.

Using the above two ranks, we can schedule and allocate registers in each region, using both a prepass and an optimized postpass as described in the previous section. However, this schedule is not profile sensitive. We next discuss Chekuri's heuristic [23] for generating a schedule that is profile-sensitive.

6.2 Chekuri's Heuristic

Let $G = (V, E)$ denote the precedence graph derived from the source program. Each vertex in the graph represents an operation τ_i with specified execution time t_i, which is the time required to execute τ_i. Each vertex also carries a weight w_i. The weight w_i is the probability that the program exits at vertex i. We are to schedule the vertices of the graph on the functional units to minimize the weighted execution time, $F = \Sigma_i w_i f_i$, where $f_i = s_i + t_i$ is the finish time of operation τ_i and s_i is its start time.

A subgraph is said to be closed under precedence if for every vertex u of the subgraph, all vertices preceding u are also in the subgraph. We define the rank of a vertex i to be the ratio $r_i = t_i/w_i$. For any set of vertices $A \subseteq V$, we define its weight as $w(A) = \Sigma_{v_i \in A} w_i$, and its execution time as $t(A) = \Sigma_{v_i \in A} t_i$; based on this the rank is $r(A) = t(A)/w(A)$. The notion of the rank of a set of vertices is meant to capture their relative importance comparing the sum of their weights to the cost of executing them. Intuitively, the sum of the weights is the contribution made by the set of vertices to the weighted finish time, while the sum of their execution times is the delay suffered by the rest of the graph as a result of scheduling the set of vertices first. A schedule for the case where the number of functional units is one is termed a sequential schedule.

Given a weighted precedence graph G, we define G^* to be the smallest precedence-closed proper subgraph of G of minimum rank. Cekuri et al. [23] prove that, for any graph G, there exists an optimal sequential schedule where the optimal schedule for G^* occurs as a segment which starts at time zero. Given G^*, we can recursively schedule G^* and the graph formed by removing G^* and putting their schedules together to obtain an optimal schedule for the entire graph. Though in general, the problem of finding G^* for an arbitrary precedence graph is NP-hard, for the case where the precedence graph is an hyperblock or superblock, G^* may be easily computed by exhaustive enumeration.

Thus, we may obtain an optimal sequential schedule for a given precedence graph G, which in turn may be used to obtain a schedule for the multiple functional unit case. Chekuri et.al. [23] also show that list scheduling using the optimal sequential schedule as the list, gives a good approximate solution for the hyperblock case. They show that, if all operations have equal execution time, the list scheduling algorithm, using the optimal sequential schedule as the list, is an approximation algorithm with a performance ratio of 2.

The modified rank of any schedule A is defined as

$$mrank(A) = \frac{\text{length of schedule for } A}{\text{sum of exit probabilities in } A}. \tag{1}$$

The intuition behind the modified rank is that the numerator is the time required to retire A, while the denominator is the benefit in retiring A. Thus, the ratio reflects the amount of computational time required per unit of exit probability. Minimizing this ratio in selecting G^* (which uses the modified rank) has the effect of maximizing the 'return on investment' in the schedule. Given a Graph G, G^* can be found by the following simple procedure.

Algorithm Finding G^* under modified rank
For each branch b of the G,
    On the given processor, construct a list schedule for subgraph G_b rooted at b, ignoring profile info
    Let T be the length of the schedule and let W be sum of exit probabilities of all exits in G_b.
    rank$(G_b) = T/W$
G^* is G_b for the earliest b in control order that has the minimum modified rank.

The heuristic converts any list scheduler for precedence graphs, to one that is sensitive to profile information for G. The heuristic takes a profile−insensitive list scheduling algorithm, and bootstraps it to be profile−sensitive. To start, the heuristic finds G^* under modified rank, using the insensitive list scheduler. It then makes the list for G^* the initial portion of the list for G. The heuristic deletes G^* from G, and iterates, appending the lists each time till all G is consumed.

Algorithm Scheduler
1. Profile-list=empty.
2. Find G^* under modified rank using the insensitive scheduler
3. Append the scheduler list of G^* to Profile-list.
4. Remove G^* from the DAG
5. If there are branches remaining, then goto Step 2.
6. List schedule using Profile-list.

6.3 Performance Evaluation

6.3.1 Machine Model

Our machine model has the following functional units:

- Integer units : 2
- Floating point units : 2
- Memory units : 2
- Branch units : 1

The latencies of operations are given below :

- Int ALU : 2
- Float ALU : 4
- Int/Float Mpy : 4
- Int/Float Divide : 12
- Load : 3
- Store : 1
- Branch : 1

6.3.2 Benchmarks

We have carried out our analysis on benchmarks available with Trimaran running on Linux 2.0.34 (Pentium 200 with 32MB). The suite contains reasonably complex benchmarks like nbradar, polyphase, wc etc. to validate our results.

program	$\alpha = 0$	$\alpha = .2$	$\alpha = .4$	$\alpha = .6$	$\alpha = .8$	$\alpha = 1$
a5	-0.3	0.2	0.2	0.2	0.2	0.7
bmm	22.3	23.1	24.1	24.2	24.2	24.2
dag	0.9	0.9	0.9	0.9	0.9	0.9
fir	6.3	6.3	6.4	6.4	6.4	6.4
hyper	3.6	3.6	3.6	3.6	3.6	3.6
idct	5.8	11.7	13.5	13.5	13.5	13.5
mm_double	16.2	19.1	19.9	19.9	19.9	19.9
nbradar	3.9	4.2	4.2	4.2	4.2	4.2
nested	0.08	0.2	0.3	0.3	0.3	0.3
polyphase	-1.5	-4.5	5.6	5.9	6.1	6.1
strcpy	0.0	0.0	0.0	0.0	0.0	0.0
wave	12.8	18.8	19.4	19.4	19.4	19.4

Table 1: Execution time % improvement of (α, β)-combined algorithm over phase ordered horizontal approach

program	$\alpha = 0$	$\alpha = .2$	$\alpha = .4$	$\alpha = .6$	$\alpha = .8$	$\alpha = 1$
a5	8.8	4.5	2.1	0.5	0.4	0.2
dag	18.2	18.9	11.1	12.1	23.7	21.8
fir	20.6	19.7	20.4	3.5	-2.3	-2.4
idct	31.1	28.2	25.8	27	21.5	21.2
nested	15.9	13.0	10.6	10.7	0.2	0.2
polyphase	6.7	0.9	-3.4	-9.1	-10.1	-3.6
wave	29.3	27.0	14.8	8.6	9.3	9.3
wc	21.0	21.0	17.2	15.1	10.8	4.3

Table 2: Execution time %improvement of profile-sensitive instruction scheduling with (α, β)-combined algorithm over phase ordered horizontal approach

6.3.3 Execution Time

We measured execution times of the code generated by the (α, β)-algorithm for values of $\alpha = 0.0, 0.2, 0.4, 0.6, 0.8, 1.0$. Table 1 gives the percentage improvement of our combined instruction scheduling and register allocation priority model over the phase ordered approach. From the table 1 we can see that we get positive performance improvement in 97% of the cases with a maximum improvement of 24.2%. We can see a performance improvement of more than 10% in more than 33% of the cases showing that the (α, β)-algorithm does a good job of incorporating register pressure sensitivity into the scheduling algorithm. The $Thick_1$ and $Thick_2$ heuristic functions have proved to be effective in reducing the overhead introduced by register allocation as well as in producing efficient schedules. Figure 10 is a %improvement graph between (α, β)-combined algorithm and phase ordered horizontal approach. This graph uses the best rank function value for each benchmark in table 1 in order.

We measured execution times of the code generated by the region based combined profile-sensitive with register pressure sensitive instruction scheduling and register allocation model over profile-insensitive horizontal model. We use critical path scheduling as profile-insensitive scheduling algorithm to derive our heuristic and compare its performance against critical path scheduling. From the table 2 we can see that we get positive performance improvement in more than 90% of the cases with a maximum improvement of 31.1%. We can see a performance improvement of more than 15% in more than 75% of the cases showing that the (α, β)-algorithm does a good job of incorporating register pressure sensitivity into the profile-sensitive instruction scheduling algorithm. The figure 11 is a %improvement graph between profile-sensitive instruction scheduling with register pressure sensitivity and phase ordered horizontal approach. This graph also uses the best rank function value for each benchmark in Table 2 in order.

The table 3 shows the percentage increase in compile time in (α, β)-combined algorithm as compared to the phase ordered horizontal approach.

program	α=0.0	0.2	0.4	0.6	0.8	1.0
a5	32.7	36.4	32.7	32.5	32.7	32.8
bmm	22.2	23.1	20.1	20.4	22.5	21.8
dag	0.0	6.3	0.0	3.9	3.2	1.6
fir	17.4	17.5	17.4	18.4	21.9	19.6
hyper	11.1	14.3	25.4	14.3	22.2	14.3
idct	18.1	17.9	14.2	22.2	16.5	16.6
mm_double	22.6	23.1	11.6	11.5	10.2	16.6
nbradar	34.4	44.3	44.3	43.1	44.5	44.3
nested	15.7	17.6	5.9	9.8	4.6	9.8
polyphase	49.6	47.9	50.8	49.5	48.7	48.8
strcpy	8.1	3.4	8.1	8.1	8.1	5.7
wave	19.7	18.1	13.7	15.9	15.9	16.7

Table 3: Compile time (percentage) increase of (α, β)-combined algorithm over phase ordered horizontal approach

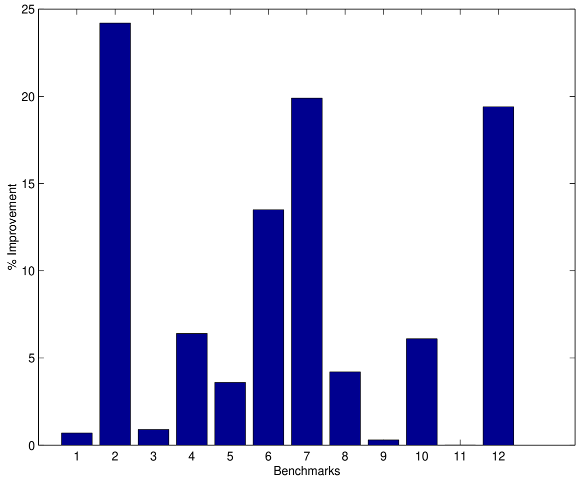

Figure 10: Execution time %improvement of (α, β)-combined algorithm over phase ordered horizontal approach

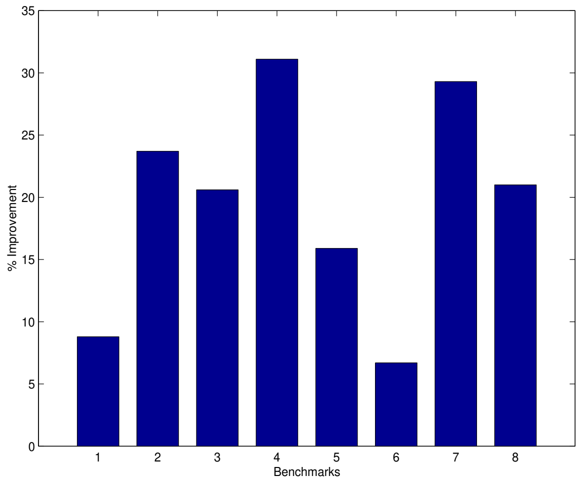

Figure 11: Execution time %improvement of profile-sensitive instruction scheduling with register pressure sensitivity over phase ordered horizontal approach

7 Conclusions and Future Work

Our results show that the combined model can give significant gains. However, the determining the values of the tuning parameters α, β is a problem. The value of α (β) may be heuristically derived for each basic block from some metric of the basic block, such as the size of the basic block or the number of live variables it contains. While the register allocator is predicate aware, the $Thick_1$ and $Thick_2$ predicates are currently not.

Another area for research is the dynamic selection of the relative coefficients for register pressure and instruction scheduling on a per region basis.

References

[1] Trimaran. *An infrastructure for compiler research in ILP.* http://www.trimaran.org.

[2] V. Kathail, M. Schlansker and B. R. Rau. *HPL-PD Architecture Specification: Version 1.1.* Technical Report HPL-93-80 (R.1). Hewlett-Packard Laboratories, July 1998.

[3] M. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik and S. G. Abraham. *Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity* . HPL Technical Report HPL-96-120. Hewlett-Packard Laboratories, February 1997.

[4] J. C. Gyllenhaal, W.-m. W. Hwu and B. R. Rau. *HMDES Version 2.0 Specification* . Technical Report IMPACT-96-3. University of Illinois at Urbana- Champaign, 1996.

[5] S. A. Gupta, V. Kathail and B. R. Rau. *Elcor's Machine Description System:Version 3.0.* HPL Technical Report HPL-98-128. Hewlett-Packard Laboratories, July 1998.

[6] B. R. Rau and J. A. Fisher. *Instruction-Level Parallel Processing: History, Overview and Perspective.* Technical Report HPL-92-132. Hewlett-Packard Laboratories, October 1992.

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

[8] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringman, 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 compilaton.* Journal of Supercomputing, Jan 1993.

[9] 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.* In Micro-25, pages 45-54, 1992.

[10] R. A. Johnson and M. S. Schlansker. *Analysis of Predicated Code.* Technical Report HPL-96-119. Hewlett-Packard Laboratories, December 1996.

[11] R. E. Hank, W. W. Hwu and B. R. Rau. *Region-based compilation: Introduction, motivation and initial experience.* International Journal of Parallel Programming 25, 2 (Apr 1997), 113-146.

[12] R. E. Hank. *Region-based compilation.* PhD. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, May 1996.

[13] A. Leung. *Global Instruction Scheduling Techniques.* Courant Institute of Mathematical Sciences, New York University, November 1997.

[14] J. R. Goodman and W. C. Hsu. *Code scheduling and register allocation in large basic blocks.* Proc. of the IEEE-ACM Supercomputing conference, 1988.

[15] S. Freudenberger and J. Ruttenberg. *Phase ordering of register allocation and instruction scheduling in Code Generaton - Concepts, Tools, Techniques.* pp.146-170, May 1992.

[16] R. Motwani, K. V. Palem, V. Sarkar and S. Reyen. *Combining register allocation and instruction scheduling.* Technical report, Courant Institute, TR 698, July 1995.

[17] S. Abraham, V. Kathail and B. Deitrich. *Meld scheduling: relaxing scheduling constraints across region boundaries.* HPL Technical Report HPL-97-39. Hewlett-Packard Laboratories, Feb 1997.

[18] F. C. Chow and J. L. Hennessy. *The prority-based coloring approach to register allocaton.* ACM Transaction on Programming Langauages and Systems, vol.12, pp.501-536, October 1990.

[19] H. Kim, K. Gopinath, V. Kathail, *Region based register allocation for EPIC processors with predication.* Parallel Computing '99, pages 36-44, 1999.

[20] M. Lam. *Software Pipelining: An effective schduling technique for VLIW machines.* Proceedings SIGPLAN'88 Symposium on Programming Language Design & Implementation, 318-328, 1988.

[21] B. Rau. *Iterative modulo scheduling: An algorithm for software pipelining loops.* Proceedings of the 27th Annual Symposium on Microarchitecture, December 1994.

[22] B. R. Rau, M. Lee, P. Tirumalai and M. S. Schlansker. *Register Allocation for software Pipelined Loops.* Proc. SIGPLAN'92 Conference on Programming Language Design and Implementation (San Francisco, June 17-19 1992).

[23] C. Chekuri, R. Motwani, R. Johnson, B.K. Natarajan, B.R. Rau, and M. Schlansker, *An Analysis of Profile-Driven Instruction Level Parallel Scheduling with Application to Super Blocks.* Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29), Paris, France, December 1996.