Journal of Instruction-Level Parallelism 6 (2004) 1-26 Submitted 10/03; published 4/04

Efficient Modeling of Itanium® Architecture during Instruction Scheduling using Extended Finite State Automata

Dong-Yuan Chen DONG-YUAN.CHEN@INTEL.COM
Lixia Liu LIXIA.LIU@INTEL.COM
Roy D.C. Ju ROY.JU@INTEL.COM
Intel Labs
Intel Corporation
Santa Clara, CA 95052 USA

Chen Fu CHENFU@CS.RUTGERS.EDU
Division of Computer and Information Sciences
Rutgers, the State University of New Jersey
Piscataway, NJ 08854 USA

Shuxin Yang SXYANG@ICT.AC.CN
Chengyong Wu CWU@ICT.AC.CN
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, P. R. China

Abstract

Effective and efficient modeling and management of hardware resources have always been critical toward generating highly efficient code in optimizing compilers. The instruction templates and dispersal rules of the Itanium® architecture add new complexity in managing resource constraints to instruction scheduler. We extended a finite state automaton (FSA) approach to efficiently manage all key resource constraints of an Itanium® architecture on-the-fly during instruction scheduling. We have fully integrated the FSA-based resource management into the instruction scheduler in the Open Research Compiler for the Itanium® architecture. Our integrated approach shows up to 12% speedup on some SPECint2000 benchmarks and 4.5% speedup on average for all SPECint2000 benchmarks on an Itanium®-based system when compares to an instruction scheduler with decoupled resource management. In the meantime, the instruction scheduling time of our approach is reduced by 4% on average.

1. Introduction

The Itanium® architecture exemplified by the Intel® Itanium® Processor Family (IPF) relies heavily on compilers to statically schedule instructions to fully utilize its wide execution resources. The majority of instruction schedulers use the dependence critical path lengths as the primary cost function to schedule instructions. Modeling of execution resources is dealt with in an ad hoc way if at all, often in a manner of scattering the hardware details across the entire scheduling phase. The Intel® Itanium® architecture [12] introduces the notion of instruction bundles and templates, which limit the instruction mixes presented to the hardware so that instructions can be dispatched efficiently to execution units. Instruction templates impose new constraints on instruction packing and dispatching, adding new complexity to resource management. In fact, such new constraints are not unique to the Itanium® architecture. Modern processors tend to specialize execution units according to functional types (e.g. memory, ALU, floating-point, etc.) and even have asymmetric functionality within the same types. A given instruction can sometimes be issued only to a particular unit within the same type, and such decisions could even be affected by surrounding instructions.

```
ld        a = [x]
add       b = y, e
ld        y = [f] ;;  // can't fit in cycle 1
ld        c = [g]
add       x = h, I
add       d = j, k ;;
```

(a) Decoupled scheduling before template selection

```
{ .mii
    ld    a = [x]              // cycle 1
    add   b = y, e             // cycle 1
    nop.i 0 ;;
} { .mii
    ld    y = [f]              // cycle 2
    nop.i 0
    nop.i 0 ;;
} { .mii
    ld    c = [g]              // cycle 3
    add   x = h, I             // cycle 3
    add   d = j, k ;;          // cycle 3
}
```

(b) Decoupled scheduling after template selection

```
{ .mii
    ld    a = [x]              // cycle 1
    add   b = y, e             // cycle 1
    add   x = h, i ;;          // cycle 1
} { .mmi
    ld    y = [f]              // cycle 2
    ld    c = [g]              // cycle 2
    add   d = j, k ;;          // cycle 2
}
```

(c) Integrated scheduling and template selection

Figure 1: An example comparing decoupled vs. integrated template selection.

A simple but sub-optimal approach to deal with these new and complex templates and instruction dispatching constraints is to add a separate instruction-packing phase after instruction scheduling. However, the example in Figure 1 argues for an intelligent scheduling phase equipped with better resource management to achieve optimal performance. Assume a hypothetical implementation of the Itanium® architecture that can issue three instructions per cycle. It has two memory (M) execution units and two ALU (I) units. For simplicity, assume there are only two instruction templates: MII and MMI, where M and I specify the functional unit types in the respective template slots. The order of slots in a template defines the sequential semantics within a cycle. The string ";;" marks a stop bit, the explicit cycle break specified by the compiler to the hardware. Anti-dependences are allowed in the same cycle as long as the instructions are ordered to reflect such dependences.

For the six instructions in Figure 1, a traditional instruction scheduler based on dependence critical paths may derive a two-cycle schedule as shown in Figure 1(a) even if it takes into account the availability of execution units. However, in the decoupled approach, a subsequent instruction-packing phase cannot find any available template to bundle the three instructions in the first schedule cycle, since they require a non-existent MIM template. Reordering the three instructions to use the MMI template is not feasible due to the anti-dependence on **y**. The bundling phase ends up forcing the "ld **y** = [f]" instruction into an extra cycle, resulting in a three-cycle schedule as shown in Figure 1(b). The bundling phase could attempt to reorder instructions beyond the current cycle with sophistication similar to instruction scheduling. This would however defeat the purpose of a decoupled approach to separate instruction scheduling from bundling for simplicity. In contrast, if template selection is integrated into the resource management of an instruction scheduler, the optimal scheduling can be achieved using the two templates MII and MMI in two cycles as shown in Figure 1(c).

Instruction scheduling is already one of the most complex phases in an optimizing compiler. Adding in the modeling of instruction templates makes it an even more challenging task. This is further complicated by the instruction dispersal rules that govern the dispatching of an instruction sequence onto execution units at each cycle, the compressed templates for packing instructions from different cycles, the one-to-many mapping possibility from instructions to functional units, etc.

The motivation of this work is to model the resource constraints due to architectural and micro-architectural specifications during instruction scheduling to generate high-performing code. Specifically, we target the constraints from the instruction dispersal rules, template selection, and execution pipelines in the Itanium® architecture, though the approach can be generalized to other processor architectures as well. The solution has to be time and space efficient to fit in a practical, production environment. We also want the modeling and management of hardware resources in an encapsulated module to allow easy migration to future implementations of the Itanium® architecture or any future processors with similar resource constraints, while the implementation of core scheduling heuristics remains independent from the micro-architectural specifications. In contrast, although producing high quality of code motivates our work, we are not looking for an optimal solution of the scheduling problem under the given resource constraints since this is an NP-hard problem and we have to develop a practical solution for a production environment.

In this work, we propose an extended finite-state automaton (FSA) to model all the resource constraints during instruction scheduling. Each state encodes the currently occupied functional units of a cycle as well as instruction templates and dispatching information. The scheduling of an instruction triggers a transition between states. Our extended FSA successfully models the new notion of instruction templates and the set of instructions dispersal rules on the Itanium® architecture with both compilation time and space efficiency. Our experimental results show that modeling these additional resource constraints is crucial toward achieving high performance on Itanium® processors. With a minimal effort we have successfully migrated the encapsulated machine model and management of hardware resources in a micro-level scheduler from the Itanium® processors to the Itanium® 2 processors under IPF, where these two generations of processors pose a number of different micro-architectural constraints.

In the rest of the paper, Section 2 provides background information and definitions of terminology. Section 3 details the functional-unit based finite-state automaton and its construction. Section 4 discusses instruction scheduling with an integrated modeling of all resource constraints based on the FSA. Section 5 presents the experimental results that compare our integrated approach with decoupled approaches. Section 6 discusses the related work, and Section 7 concludes this paper.

2. Background

2.1. Intel® Itanium® architecture

The Itanium® architecture [12] uses wide instruction words as in the Very Long Instruction Word (VLIW) architecture. There are four *functional unit types* – M (memory), I (integer), F (floating point), and B (branch). Each instruction also has an *instruction type* – M, I, A, F, B, and L. The instruction type specifies the functional unit type where an instruction can be executed, where instruction type A (i.e., ALU instructions) can be dispatched to functional units of either type M or I, and instruction type L consumes one I and one F units. Instructions are encoded into bundles where each *bundle* contains three instructions with a specific instruction template. Each instruction occupies one slot in a bundle. A *template* specifies the functional unit type for each contained instruction. There are 12 basic templates, such as MII, MMI, MFI, MIB, etc. A *stop bit* indicates to the processor that the instructions before and after the stop bit may have certain resource dependencies and are to be executed at different cycles. Each basic template type has two versions: one with a stop bit after the third slot and one without. Two of the basic templates, MI_I and M_MI, have a stop bit in the middle of a bundle. For example, the MI_I template has a stop bit between the two I slots. We call the two *compressed templates* because they allow the packing of instructions from different cycles into smaller code size. Flow and output register dependences are generally disallowed (with a few exceptions) among the same groups of instructions delimited by explicit stop bits, but register anti-dependences are generally allowed.

Each implementation of the Itanium® architecture has its own micro-architectural features. For example, the Itanium® processor [13] can issue and execute up to six instructions (two bundles) per cycle. It has a total of 9 *functional units* (FU): 2 M-units (M0 and M1), 2 I-units (I0 and I1), 2 F-units (F0 and F1), and 3 B-units (B0, B1, B2). Functional units under the same type may be asymmetric, requiring certain instructions to be executed only to a particular FU of a FU type. Each processor generation may also have different instruction latencies and execution-pipeline bypasses, resulting in varying latencies between the same pair of dependent instructions when they are dispatched to different FUs. Each processor generation has its own set of *instruction dispersal rules* that describes how each instruction is dispersed to a FU in an instruction sequence. Depending on the bundle location (the first or second in a cycle) and slot location in an instruction fetch cycle, the same instruction may be dispersed to different FUs. How one instruction is ordered or aligned could force another instruction intended for the same cycle to be stalled to a later cycle due to conflict in critical FUs. On the Itanium® processor, in a cycle started with an MII bundle, the instruction in the first I slot always goes to the I0 unit. If the instruction in the second I slot can be executed only on the I0 unit, which is already taken, the processor will force it to execute in the next cycle. A detailed description of all of these micro-architectural features for the Itanium® processor can be found in [13].

2.2. High- and micro-level instruction scheduling

Our integrated instruction scheduler divides the task of instruction scheduling into two – the high-level instruction scheduling and the micro-level scheduling. This integrated instruction scheduler has been implemented in the Open Research Compiler. Depending on the scheduling scope, the scheduler can perform global scheduling or basic block local scheduling. The high-level instruction scheduling algorithm in our global scheduler is based on [2] but performs on the scope of single-entry-multiple-exit regions containing multiple basic blocks with internal control flow transfers. Note that basic block is a degenerated region. Within each region, a dependence DAG is constructed, where each node represents an instruction in the region and each edge indicates a dependence relation (flow, output, or anti) between the instructions on the two ends of the edge. The edges are weighted, and the weight is the minimal latency required by the hardware to issue the two instructions. The implementation is a forward, cycle scheduling. The enhanced priority function is based on the path lengths (to the last cycle of the region) weighted by execution frequency in the global dependence DAG. Each basic block in the region is selected as the target basic block in a topological order weighted with execution frequencies. If all of the predecessors of an instruction have been scheduled and the scheduling latency on each incoming dependence edge is satisfied, this instruction is ready to schedule. All ready instructions from the basic blocks dominated by the target basic block are the candidates to be scheduled into the target basic block. The high-level instruction scheduling determines the issue cycle of each instruction according to dependences and instruction execution latencies. These scheduling steps are repeated for every basic block to serve as a target basic block and terminated when all of the instructions in the region are scheduled. When one region is scheduled, the next most frequently executed region is then scheduled.

The global instruction scheduling also drives a number of machine-dependent optimizations to fully utilize the architectural features. These include control and data speculation to move load instructions across branches and aliasing stores. To support speculation, a new type of speculative edge is introduced in the dependence DAG. In contrast to a normal dependence edge whose dependence must be observed during the scheduling sequence, a speculative edge indicates that the represented dependence may be ignored due to the special hardware support for control and data speculation on Itanium® architecture. While selecting ready instructions, only normal dependence edges are examined. When an instruction is scheduled, we then examine whether the dependence on any speculative edge to this instruction is not satisfied. If so, this instruction is scheduled speculatively and has to be marked so with proper encoding to use the speculative architecture support. The global instruction scheduling phase can also schedule instructions across joint points with compensation code generated and can schedule partially ready instructions. The high-level instruction scheduling in the local scheduler operates on a basic block scope without speculation. Both the global and local instruction schedulers incorporate resource management thru the same micro-level scheduler.

The micro-level scheduling takes care of the placement of instructions and resource management within a cycle. It can reorder instructions within the same cycle in order to achieve the best FU assignment and template selection, as long as it honors all the instruction dependencies specified by the high-level scheduler. The micro-level scheduling thus shields from high-level scheduling algorithm the complicated resource management of the Itanium® architecture, such as available FUs, instruction dispersal rules, and instruction bundles and templates.

2.3. Building machine information into compiler

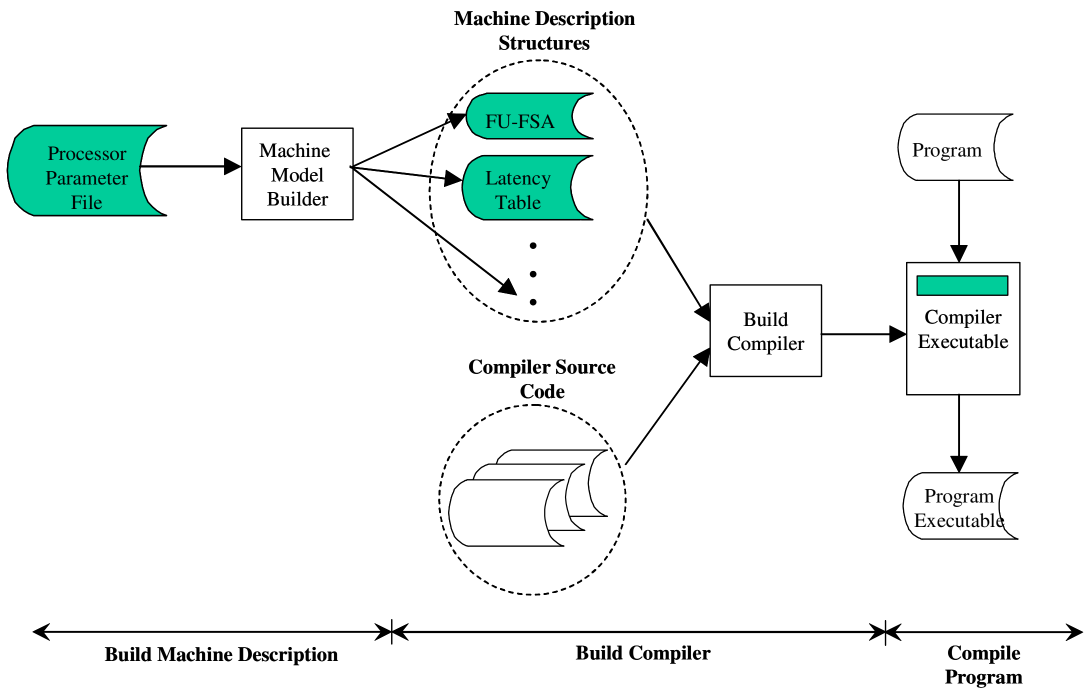


Figure 2: Building and propagating machine information.

Figure 2 shows the creation and usage of the machine-specific information in various stages of the compiler life cycle. A Machine Model Builder (MM Builder) constructs a set of machine description structures from a published processor parameter file [14] in the compiler building process. The processor parameter file describes the micro-architecture details of an Itanium® processor, including available functional units, instructions mapping to functional units, latency, etc. The generated machine description structures include tables that specify the numbers and types of machine resources (such as machine width, functional units, and templates), the instruction latency, the pipeline bypass constraints, etc. Our functional-unit based FSA, to be discussed in Section 3, is also constructed by the MM builder. The functional-unit based FSA, as part of the machine description structures, contains a set of valid FSA states and a state transition table. It also incorporates valid template combinations and instruction dispersal rules for each state.

The machine description structures become part of the static data in the compiler executable after the compiler is built. Both the high-level and micro-level instruction schedulers retrieve machine-specific information from the machine description structures.

3. Functional-unit based finite-state automata

3.1. Motivation of functional-unit based finite-state automata

Finite state automata have been used in several modern instruction schedulers to model the resource contention in execution pipelines [1, 21]. A well-designed FSA often results in a simple and compact implementation and exhibits better efficiency in both space and time.

While the pipeline resource contention of Itanium® processors is simple to model, the instruction templates and dispersal rules introduce new challenges. The latency between a pair of dependent instructions depends not only on the type of source instruction but also on the functional unit where it is executed. The bundle template and the slot location in a bundle determine the functional unit assigned to execute an instruction. Hence selecting the instruction template and placing instructions into the proper slot locations in a bundle is critical toward achieving a high quality schedule.

The importance of template selection intuitively suggests a template-centric model for managing resources. When an instruction is being scheduled, a template-centric model first decides the template assignment for the current schedule cycle and the slot location for the instruction. The instruction latency and the executing functional unit of the instruction are then derived from the instruction template and slot location assigned per instruction dispersal rules. All possible template assignments are dynamically enumerated at each scheduling attempt to select the template and slot.

To improve the compile-time overhead in dynamic template-assignments enumeration, one could build a template-based FSA off-line to guide the selection of template assignments. The template-based FSA would model the template assignment and slot usage of an execution cycle. Each state in a template-based FSA would record all the possible selections of instruction templates under certain slot usage in a single execution cycle. When an instruction was scheduled, an unused slot S would be picked for the instruction and the FSA would transit to the next state where the slot S becomes taken. Template assignment would be selected from the set of templates in the state. Size is one major problem in such a template-based FSA. For the two-bundle wide Itanium® processor, there are at least 68 possible template assignments in a cycle, after accounting for the availability of functional units. The theoretical upper bound on the number of states in a template-based FSA is 2^{68}, the size of the power set of all possible template assignments. Even with aggressive trimming, a template-based FSA still needs dozens of thousand states.

To achieve efficiency in both space and time, we take a functional-unit-centric approach. At the core is a Functional-Unit-based FSA (or FU-FSA in short). Each state in the FU-FSA represents the set of functional units that are in use (FU usage set) in a schedule cycle. Instructions scheduled into a schedule cycle are treated as a sequence of FU-FSA state transitions. Instruction templates and dispersal rules are modeled in each state by a list of legal template assignments of the state. A template assignment is legal for a state if its FU usage set is a superset of the FU usage set of the state. Only states with at least one legal template assignment are included in the FU-FSA.

The FU-FSA is much more space efficient than the template-based FSA. The FU-FSA for the nine-FU Itanium® processor has at most 2^9 or 512 states, much smaller than that of a template-based FSA. After excluding states with no legal template assignment, the FU-FSA for an Itanium® processor contains 235 states. Each state has no more than 38 legal template assignments, while 75% of the states have less than 10 legal template assignments. Therefore the FU-FSA is very compact in term of memory usage and is highly scalable to a wider machine.

3.2. Off-line construction of FU-FSA

The FU-FSA is constructed to represent all valid FU usage patterns as the states of the FU-FSA. The construction of FU-FSA further preprocesses the constraints imposed by instruction templates and instruction dispersal rules of the Itanium® processor off-line when the compiler is built. These constraints are incorporated into the FU resource modeling of each state through a list of legal template assignments associated with each state. The construction of FU-FSA ensures that the FU resource usage represented by each state can only be assigned template that is in its list of legal template assignments. The online overhead for checking the instruction template and dispersal constraints is thus significantly reduced.

```
BuildFSA() {
    FOREACH FU usage set PV DO {
        FOREACH template assignment T with at most 2 bundles DO {
            TV = FU usage set of T;
            IF (PV is a subset of TV) {
                IF (PV is not in FSA) {
                    Add PV as a new FSA state;
                }
                Add T to FSA[PV].TAs;
            }
        }
        IF (PV is in FSA) {
            Sort FSA[PV].TAs according to priority criteria;
        }
    }
    Build FSA transition table;
}
```

Figure 3: Pseudo code for building an FU-FSA for an Itanium® processor.

Figure 3 outlines the algorithm for constructing the FU-FSA for the two-bundle wide Itanium® processor. It can be easily extended to generate a FU-FSA for any Itanium® processor that can issue N bundles per cycle. The algorithm enumerates all possible FU usage patterns, represented as a FU usage set *PV*. For each *PV*, we scan all possible template assignments of up to 2 bundles. If a template assignment *T* has a FU usage set *TV* per instruction dispersal rules and *TV* is a superset of *PV*, *T* is a legal template assignment for *PV*. In this case, *PV* is added to the set of FU-FSA states. The template assignment *T* is also added to the list of legal template assignments (TAs) for the state *PV*, that is, *FSA[PV]*.

After all template assignments for the FSA state with the FU usage set *PV* have been enumerated and examined, the list of legal template assignments for *PV* is sorted according to certain priority functions. The priority functions are chosen based on the way the FU-FSA is used online and its effect on the generated code quality. For instance, suppose we have a set of instructions that are scheduled to the same cycle and results in the FU usage set of the FSA state *S*. We want to select the template assignment for state *S* after scheduling for the cycle is completed. We can scan the list of legal template assignments of state *S* and use the very first one that meets all required dependence constraints from the set of instructions. In order to minimize code size for Itanium® processors, we sort the list of template assignments according to the following two priorities:

1. Ascending bundle count.
2. Putting template assignments with compressed template(s) ahead of full template(s).

The first priority favors the usage of a single-bundle template assignment over a two-bundle template assignment whenever the dependence constraints are met. The second priority prefers the usage of compressed template(s) over full template(s) in packing instruction into templates. The two priority functions combined give a template assignment that has the smallest code size whenever possible.

Finally the FU-FSA transition table is constructed after all legal states of FU-FSA are included. A state of the FU-FSA represents the set of functional units that are in use (FU usage set). And the transition edge from one state to another is annotated with the FU resource that are consumed (or released).

3.3. On-line usage of FU-FSA

The FU-FSA is used to model the resource usage and constraints of a single execution cycle. The schedule of a region is logically organized as a sequence of schedule cycles, with each schedule cycle modeled by one FU-FSA. When an instruction is scheduled into a cycle, it leads to a transition of the FU-FSA that is associated with the schedule cycle.

Figure 4 illustrates a simple sequence of FU-FSA state transitions for a schedule cycle where the FU usage set in each state is represented as a bit vector. The FU-FSA starts out with the initial state 0x00, indicating no FU is used at the beginning. When a FU is assigned to an instruction during instruction scheduling, the current state follows the transition edge marked with the consumed FU and transits to a new state. The new state has the bit corresponding to the newly occupied FU set. For example, if we start with the current state 0x00 and add a new instruction that occupies the M0 unit (bit 0 in the bit vector), the new state will be 0x01. Figure 4 highlights the sequence of FU-FSA transitions, following the transition path M1→I0→I1→F0 that leads from the initial state 0x00 to the state 0x1E. It also shows the list of legal template assignments for the state 0x1E, assuming the target is an Itanium® processor.

When the high-level scheduler requests an instruction to be scheduled at a cycle, the micro-level scheduler needs to find a FU *P* for the instruction at the current schedule cycle that can lead to a legal transition in the FU-FSA. A legal FU-FSA state transition needs to satisfy two conditions. First, there must be a transition edge annotated with *P* out of the current state. This basically checks for the availability of FU resource *P*. The second condition requires that at least one legal template assignment under the new state satisfies the dependence constraints for all instructions currently scheduled in the cycle. Note that, in general, the Itanium® architecture allows concurrent execution in the same cycle of instructions with register write-after-read dependences and memory read-after-write, write-after-write and write-after-read dependences. We refer to such dependence constraints among instructions in the same scheduled cycle as *intra-cycle instruction dependences*. Dependence constraints that span multiple schedule cycles are called *inter-cycle instruction dependences*.

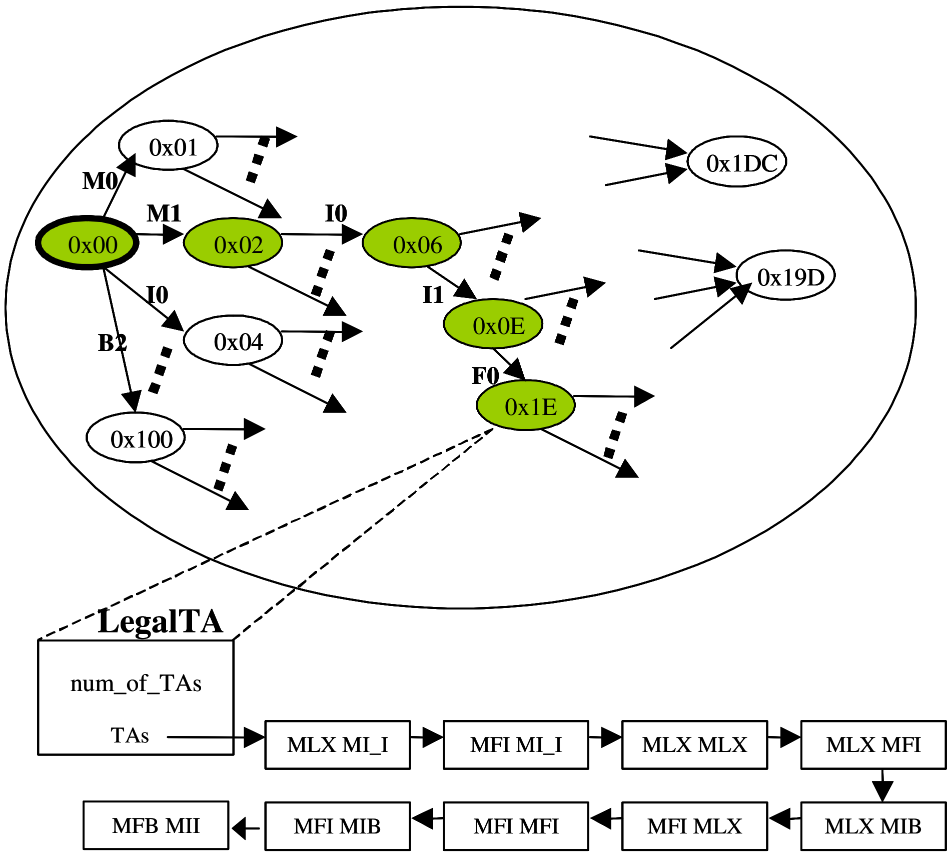


Figure 4: A state transition example in a functional-unit based FSA.

When there is no intra-cycle instruction dependence, no check is required for the second condition since the construction of FU-FSA guarantees that at least one legal template assignment exists for each state. When there are intra-cycle dependences in the schedule cycle, the FU-FSA transition needs to ensure the existence of at least one template assignment that can lay out instructions in the required dependence order. This is accomplished by scanning the list of legal template assignments of the new FU-FSA state. In either case, the final selection of template assignment is needed only when scheduling for a cycle has completed, instead of done at every scheduling attempt.

As illustrated in Figure 4, each state has a structure *LegalTA* that contains the list of legal template assignments (TAs) for that state per instruction dispersal rules. When instruction scheduling for a cycle has completed and it is time to finalize the template selection for the cycle, the list of legal template assignments of the current FU-FSA state is scanned to find a template assignment that can best realize the set of instructions in the cycle. One simple approach is to pick the first template assignment in the list that satisfies all required constraints. By properly arranging the list of template assignments for each state during the FU-FSA construction as discussed in Section 3.2, the simple approach can optimally select the template assignment that lead to smaller code size (or bundle count). A smaller code size in general leads to better performance due to a reduction in I-cache misses.

4. Instruction scheduling with integrated resource management

4.1. Scalar instruction scheduling using FU-FSA

With a FU-FSA based micro-level scheduler taking care of the resource management, the instruction scheduler can focus on high-level scheduling decision. Figure 5 highlights the interaction between the high-level instruction scheduler and the micro-level scheduler. On the left-hand side is the flow of a typical instruction scheduler. The right-hand side shows the functions done in the micro-level scheduler.

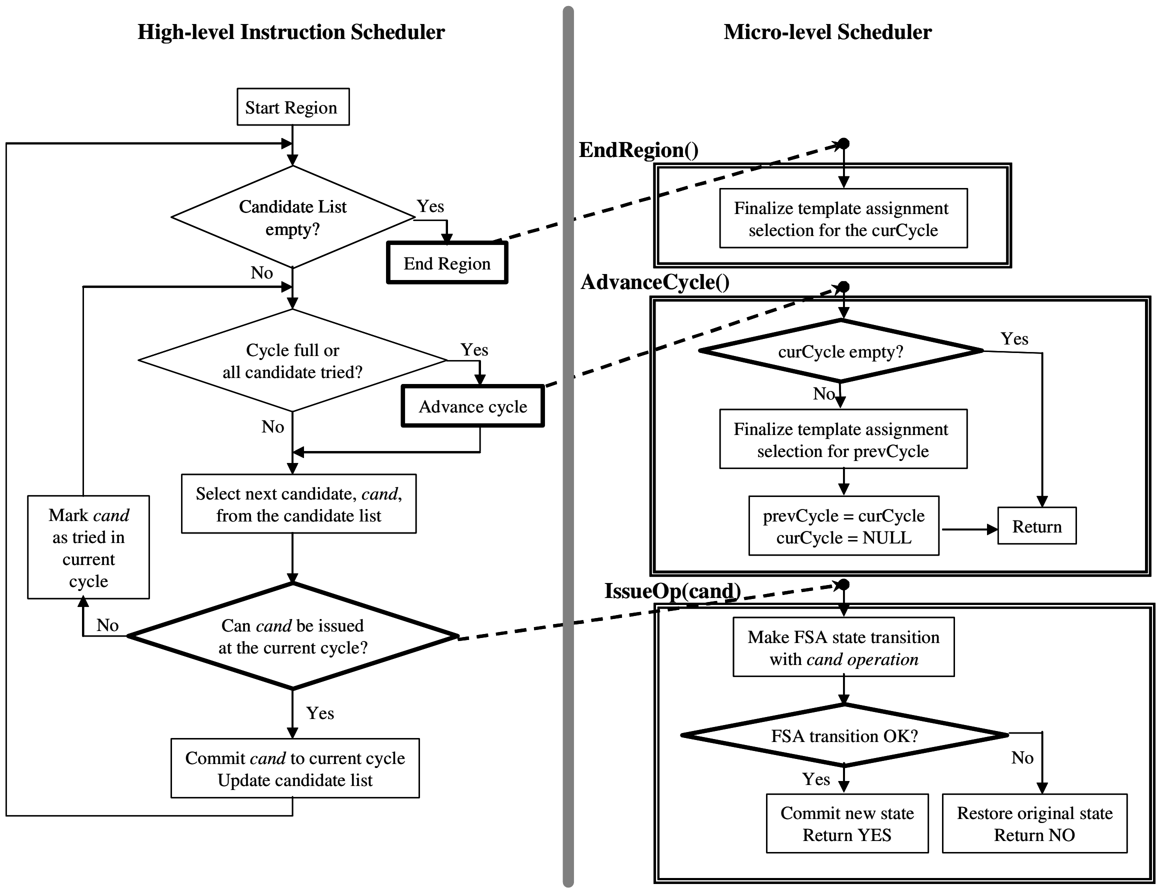

High-level Instruction Scheduler
Micro-level Scheduler
Start Region
Candidate List empty?
Yes
End Region
No
Cycle full or all candidate tried?
Yes
Advance cycle
No
Select next candidate, *cand*, from the candidate list
Mark *cand* as tried in current cycle
No
Can *cand* be issued at the current cycle?
Yes
Commit *cand* to current cycle Update candidate list
EndRegion()
Finalize template assignment selection for the curCycle
AdvanceCycle()
curCycle empty?
Yes
No
Finalize template assignment selection for prevCycle
prevCycle = curCycle curCycle = NULL
Return
IssueOp(cand)
Make FSA state transition with *cand operation*
FSA transition OK?
No
Yes
Commit new state Return YES
Restore original state Return NO

Figure 5: Interaction between high-level and micro-level instruction scheduling.

The instruction scheduler repeatedly picks the best candidate from a list of instructions in the schedule region that are ready for execution at the current cycle. The schedule region could be a simple basic block in a basic block scheduler. Or it could be the single-entry-multiple-exits region used in the global instruction scheduler of ORC. The instruction scheduler then consults the micro-level scheduler through the *IssueOp* function to check for resource availability. The *IssueOp* function determines whether there is a FU available for the candidate instruction and whether there exists a legal template assignment that satisfies instruction dependence constraints if exist. Using the FU-FSA, it simply picks an available FU for the candidate instruction and sees whether the FU results in a legal FU-FSA state transition.

If *IssueOp* completes successfully, the candidate instruction is committed to the cycle. If *IssueOp* is not able to fit the instruction in the current cycle, the instruction scheduler marks the candidate instruction as being tried already. In either case, the scheduler picks another candidate instruction from the candidate list to schedule until the current cycle is full or there is no more candidate instruction to try. The instruction scheduler then closes the current cycle and advances to the next cycle.

The instruction scheduler now needs to decide when to finalize the template assignment for each cycle. One alternative is to finalize the template assignment on-the-fly as soon as scheduling for a cycle is completed (the 1-cycle template selection heuristic). The template selection algorithm looks at the one-cycle window in making template assignment. Or one may defer the template assignment until the whole schedule region is completely scheduled. Selecting the template assignments for several cycles at once allow better packing of instructions in adjacent cycles by exploiting compressed templates. However, it requires extra compilation time and a larger space for maintaining the states of several cycles.

Our instruction schedulers employ a 2-cycle template selection heuristic that is able to utilize compressed templates for better instruction packing while incurs minimal compilation time and space overhead. Instead of finalizing the template assignment as soon as the scheduling of a cycle is done, the template assignment is selected with a one-cycle delay to give a window of two schedule cycles for template selection. Only the states of two cycles need to be maintained during instruction scheduling, namely, the previous cycle (*prevCycle*) and the current cycle (*curCycle*). When the high-level scheduler advances a cycle, the *AdvanceCycle* function in the micro-level scheduler is invoked to finalize the template assignment for the previous cycle. By looking at both previous and current cycles when selecting the final template assignment of the previous cycle, it allows the utilization of compressed templates. It results in a good balance between the quality of generated code and the space and time efficiency of the instruction scheduler.

Once scheduling for the schedule region is completed, the *EndRegion* function in the micro-scheduler is called to finalize the template assignment of both the previous and current cycles.

The pseudo code in Figure 6 illustrates a simplified implementation of *IssueOp* using the FU-FSA. Inputs to the *IssueOp* function include the instruction ***op*** to be scheduled and the FU-FSA state of the schedule cycle it is scheduled into. The *IssueOp* function also has access to all the dependence edges involving instruction ***op*** in the global dependence DAG built for the schedule region. The simplified *IssueOp* function assumes that the latency requirements between op and other instructions due to inter-cycle dependences have been verified.

```
IssueOp(op, cycle) {
    funcUnits = FUs op can be issued to;
    freeUnits = unoccupied FUs in cycle;

    // Try available FUs first.
    candidateUnits = funcUnits & freeUnits;
    FOREACH FU in candidateUnits DO {
        Record op issued to FU in cycle;
        state = getFSAState(cycle);
        IF (state is valid) {
            IF (intra-cycle dependence in cycle) {
                FOREACH ta in FSA[state].TAs DO {
                    IF (ChkCycleDep(cycle,ta) == TRUE)
                        RETURN YES;     // Succeed
                }
            } ELSE
                RETURN YES;             // Succeed
        }
        Back out op from FU in cycle;
    }

    // Try permuting FU assignments.
    candidateUnits = funcUnits & ~freeUnits;
    FOREACH FU in candidateUnits DO {
        IF (FU is locked) CONTINUE;
        old_op = cycle->op_in_FU(FU);
        Back out old_op from FU in cycle;
        Issue and lock op to FU in cycle;
        IF (IssueOp(old_op, cycle) == TRUE)
            RETURN YES;                 // Succeed
        Back out op from FU in cycle;
        Record old_op issued to FU in cycle;
    }
    RETURN NO;      // Fail
}
```

Figure 6: Pseudo code for IssueOp using FU-FSA.

Inside the *IssueOp* function, unoccupied FUs are first selected for the new instruction ***op***, as shown in the first FOREACH loop over the candidate FUs. When a tentative FU is selected for ***op***, the cycle is updated to reflect the assignment. The new FU-FSA state is checked to make sure it is a legal state. Furthermore, if there are dependences among instructions within the cycle, the list of legal template assignments (TAs) for the new state must be examined to ensure at least one legal template assignment exists for the required instruction sequence in the cycle. The *ChkCycleDep* function performs the dependence check given a template assignment and the instructions scheduled in the cycle with their FU assignments.

If op is not able to use any of the unoccupied FUs, occupied but valid FUs for op are tried next. *IssueOp* will re-arrange the FU assignments for instructions already scheduled in the current cycle to exploit the best resource usage. It involves backing out the FU assignment for one or more instructions and re-arranges the mapping of instructions to FUs. The second FOREACH loop over the candidate FUs performs this FU re-mapping. A FU locking mechanism avoids repeating previously tried combinations to ensure termination of the algorithm. Heuristics that give higher priority to the most constrained instructions can be applied to reduce the search space during FU re-mapping.

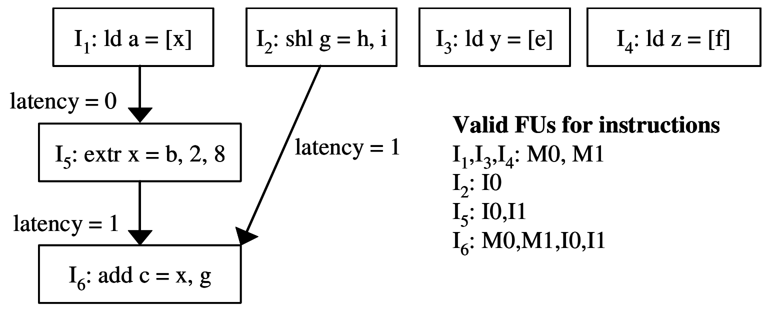
4.1.1. A scalar instruction scheduling example

Figure 7 shows an example of the high-level scheduler and the micro-level scheduler in action. This example assumes that the schedule region is a simple basic block. Figure 7(a) shows the dependence DAG of the instructions in the schedule region. It also lists the FUs that are valid for each of the instructions.

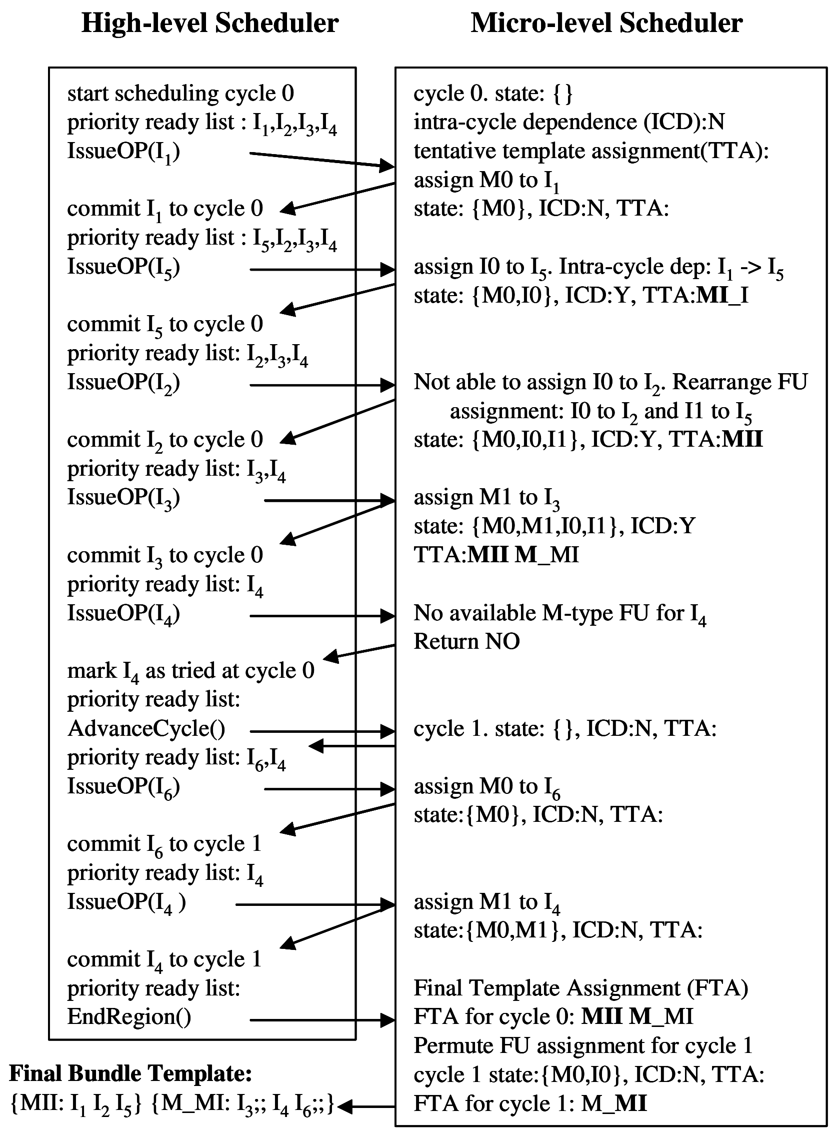
Figure 7(b) demonstrates the step-by-step interaction and the internal states of the high-level scheduler and the micro-level scheduler. The high-level scheduler builds a ready list of candidate instructions based on their respective path length in the DAG. In case of a tie, the order is arbitrary. In the micro-scheduler, it shows the FU-FSA state for the current schedule cycle and the tentative template assignment (TTA) if an intra-cycle dependence (ICD) exists in the cycle.

The high-level instruction scheduler starts with scheduling instruction I_1 at cycle 0, which is assigned to FU M0 by the micro-level scheduler. Next come instruction I_5 that must be executed after I_1 because of the anti-dependence on register x. Instructions I_1 and I_5 can be executed at the same cycle as long as I_1 is put before I_5. The micro-scheduler picks FU I0 for I_5 and makes sure that the template **MI**_I does allow FU M0 to goes before FU I0. Instruction I_2 is scheduled next. Note that I_2 can only be executed on FU I0, which is already taken by I_5 at cycle 0. However, the micro-level scheduler is able to reassign I_5 to FU I1 and make FU I0 available for I_2. The FU rearrangement maintains the ordering between I_1 and I_5 since FU M0 goes before FU I1 in the **MII** template. The scheduling of instructions continues until all instructions in the basic block are scheduled in two schedule cycles.

When the high-level instruction scheduler completes all instructions in the basic block, it invokes the *EndRegion* function and the micro-level scheduler proceeds to finalize the template assignment for both cycles. During the final selection of template assignment, the FU assignment in cycle 1 is rearranged from {M0, M1} to {M0, I0} to enable the use of M_**MI** template assignment. The reassignment of FUs can be accomplished using a process similar to that in the *IssueOp*. The final schedule results in two bundles of instructions with an M_MI compressed template in the second bundle.



(a) Dependence DAG for the sample instructions.


High-level Scheduler
Micro-level Scheduler
start scheduling cycle 0
priority ready list : I_1,I_2,I_3,I_4
IssueOP(I_1)
cycle 0. state: { }
intra-cycle dependence (ICD):N
tentative template assignment(TTA):
assign M0 to I_1
state: {M0}, ICD:N, TTA:
commit I_1 to cycle 0
priority ready list : I_5,I_2,I_3,I_4
IssueOP(I_5)
assign I0 to I_5. Intra-cycle dep: I_1 -> I_5
state: {M0,I0}, ICD:Y, TTA:**MI**_I
commit I_5 to cycle 0
priority ready list: I_2,I_3,I_4
IssueOP(I_2)
Not able to assign I0 to I_2. Rearrange FU assignment: I0 to I_2 and I1 to I_5
state: {M0,I0,I1}, ICD:Y, TTA:**MII**
commit I_2 to cycle 0
priority ready list: I_3,I_4
IssueOP(I_3)
assign M1 to I_3
state: {M0,M1,I0,I1}, ICD:Y
TTA:**MII** **M**_MI
commit I_3 to cycle 0
priority ready list: I_4
IssueOP(I_4)
No available M-type FU for I_4
Return NO
mark I_4 as tried at cycle 0
priority ready list:
AdvanceCycle()
cycle 1. state: { }, ICD:N, TTA:
priority ready list: I_6,I_4
IssueOP(I_6)
assign M0 to I_6
state:{M0}, ICD:N, TTA:
commit I_6 to cycle 1
priority ready list: I_4
IssueOP(I_4)
assign M1 to I_4
state:{M0,M1}, ICD:N, TTA:
commit I_4 to cycle 1
priority ready list:
EndRegion()
Final Template Assignment (FTA)
FTA for cycle 0: **MII** **M**_MI
Permute FU assignment for cycle 1
cycle 1 state:{M0,I0}, ICD:N, TTA:
FTA for cycle 1: M_**MI**
Final Bundle Template:
{MII: I_1 I_2 I_5} {M_MI: I_3;; I_4 I_6;;}

(b) Scheduling states in high-level and micro-level schedulers.

Figure 7: A simple scalar instruction scheduling example.

4.2. Software pipeline scheduling using FU-FSA

The FU-FSA model can be easily integrated into a software-pipelining scheduler as well. The FU-FSA-based micro-level scheduler works on cycle-level resource management. It coordinates with the high-level scheduler to obtain dependence information among instructions and to compute the latency between pairs of dependent instructions.

To simplify our discussion, let us assume the software pipelining scheduler is a modulo scheduler [24]. In contrast to a scalar instruction scheduler that generally considers only loop-independent dependences, a software pipeline scheduler needs to consider both loop-independent dependences and loop-carried dependences. Thus both types of dependences must be examined when the micro-level scheduler looks at the intra-cycle instruction dependences that constrain the ordering of instructions during template selection. Furthermore, the micro-level scheduler must model a modulo resource table that is logically laid out as *N* consecutive scheduling cycles, where *N* is the Initiation Interval of the pipelined schedule under construction. In the modulo resource table, schedule cycles *(c+k*N)*, for $k \geq 0$, are all modeled by the FU-FSA associated with modulo schedule cycle *c*. Finally, the final template assignments for the *N* modulo schedule cycles can only be decided and selected after the software pipeline scheduler have completed the construction of the final pipelined schedule.

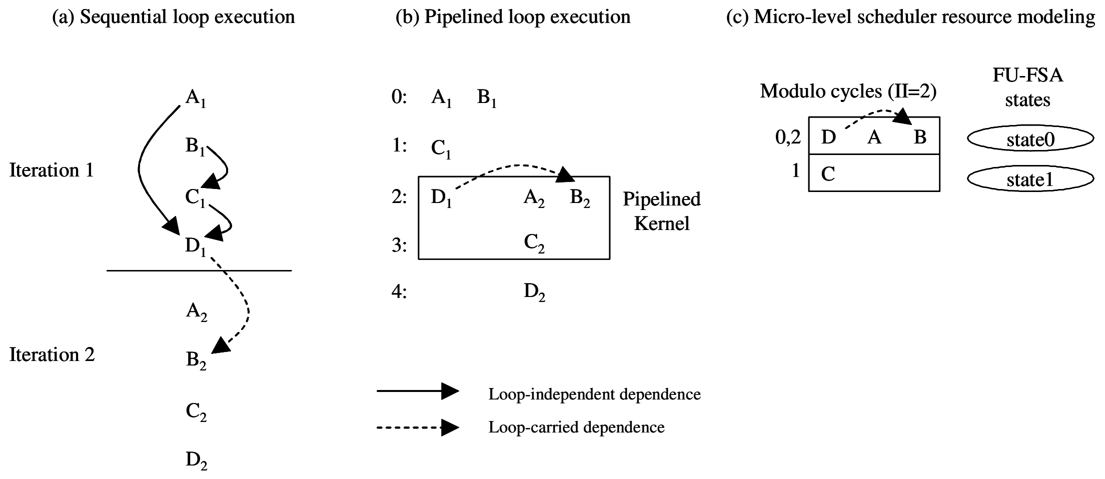


Figure 8: Modeling resources using FU-FSA for software pipelining.

Figure 8 illustrates how these additional factors are considered when the FU-FSA based micro-scheduler is working with a modulo scheduler. Figure 8(a) shows the first two iterations of a loop with four instructions A, B, C and D, annotated with subscripts indicating the loop iteration they come from. Figure 8(b) shows the execution of the two loop iterations after software pipelined with an Initiation Interval (II) of 2. Figure 8(c) shows the modulo resource table that must be modeled in the micro-level scheduler. In this example, instructions A and B are scheduled to cycle 0 and instruction D is scheduled to cycle 2 in the pipelined schedule. However, instructions A, B and D all consume resources in modulo schedule cycle 0 and trigger transitions in the FU-FSA state corresponding to modulo schedule cycle 0. Furthermore, while examining the intra-cycle dependences of modulo schedule cycle 0, the loop-carried dependence from instruction D of iteration 1 to instruction B of iteration 2 must be considered. Hence we must ensure instruction D comes before instruction B in selecting template assignment for module schedule cycle 0.

5. Experimental results

The proposed instruction scheduling integrated with FU-FSA-based resource management for the Itanium® architecture has been fully implemented in the Open Research Compiler (ORC) for Itanium® architecture [20]. We compare our integrated approach with an approach of decoupled instruction scheduling and template selection in terms of run-time and compilation-time performance.

ORC includes advanced program optimizations, such as inter-procedural analysis and optimizations, loop-nest transformations, machine-independent optimizations, and profile-guided optimizations and scheduling. The global instruction scheduling reorders instructions across basic blocks. There has been a large amount of research work done on instruction scheduler [2, 3, 8, 10, 16, 17, 18, 23]. Our global instruction scheduler uses a forward, cycle scheduling algorithm based on [2], but it performs on the scope of single-entry-multiple-exit regions as described in Sec 2.2. The global instruction scheduling also utilizes the special architectural features of Itanium® processors and performs control and data speculation to move load instructions across branches and potential aliasing stores. In case there are spills from register allocation, the local scheduling is invoked for the affected basic blocks. The local scheduling operates on a basic block scope without speculation. Both the global and local instruction scheduling incorporate our FU-FSA based resource management.

We compare two levels of integration in resource management and instruction scheduling for Itanium® processors, namely, the decoupled bundling approach (BASE) and our integrated FU-FSA resource modeling approach (FUFSA). Both BASE and FUFSA use the same instruction scheduler and heuristics. In the BASE configuration, the instruction scheduler performs scheduling based on instruction dependences, instruction latency, pipeline bypass constraints, issue width, and the functional unit availability. But instruction dispersal rules and templates are only taken into account during a subsequent, independent bundling phase that is dedicated to pack instructions under templates. The independent bundling phase invoked after the local scheduling uses the same FU-FSA-based machine model and micro-level scheduler in much the same way as in FUFSA. The bundling phase is not allowed to reorder instructions across the cycle boundary marked by the upstream schedulers.

In the FUFSA configuration, the instruction scheduler incorporates the FU-FSA-based resource management that accounts for instruction latency, pipeline bypass constraints, issue width, types of FUs, templates, and dispersal rules. Instructions are packed under templates on-the-fly when instructions are scheduled without a separate bundling phase.

For each of the BASE and FUFSA configurations, we collected data on two bundling heuristics. The first heuristic (1-cycle template selection) selects the template for instructions in a cycle as soon as scheduling to the cycle is done. It effectively ignores the two compressed templates, MI_I and M_MI. These configurations are called BASE-1 and FUFSA-1 respectively. The second heuristic (2-cycle template selection) defers the template selection of a completed cycle until the next schedule cycle is done, enabling the use of compressed templates to reduce code size. We called these configurations BASE-2 and FUFSA-2. Note FUFSA-2 is the target approach of this work as described in the preceding sections.

We measured performance using all 12 SPECint2000 benchmark programs with full reference input sets. These benchmark programs are compiled using ORC with the peak performance options, which include inter-procedural optimizations, function inlining, profile feedback, and extensive intra-procedural and Itanium® processor specific optimizations. The generated codes are run and measured on a 733 MHz Itanium®-based workstation with 2 MB L3 cache and 1 GB memory, running RedHat 7.1 version of Linux.

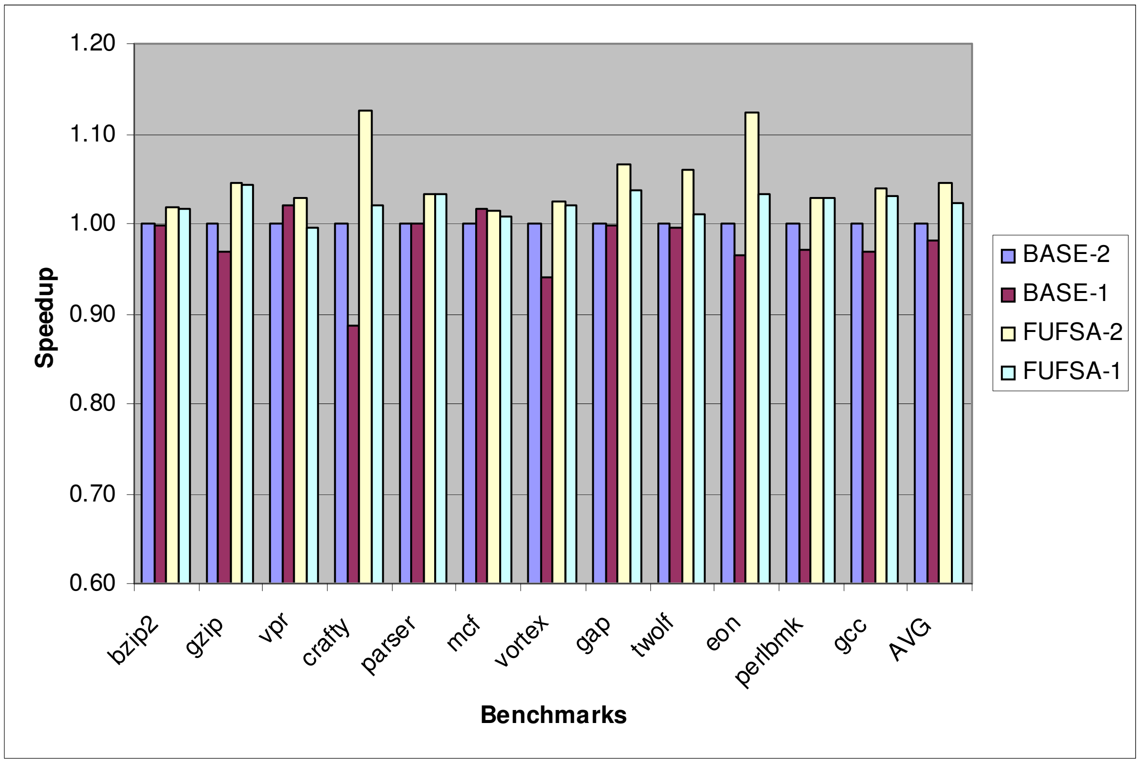


Figure 9: CPU cycles speedup over BASE-2.

Figure 9 shows the speedup in CPU cycles of all configurations over BASE-2. FUFSA-2 outperforms BASE-2 on all benchmarks, with an average of 4.5% speedup. Crafty and Eon show an impressive speedup of over 12%. We also observed that in general the 2-cycle template selection heuristic performs better than the 1-cycle template selection heuristic. On average, BASE-2 gets a 1.8% speedup over BASE-1 and FUFSA-2 obtains a 2.26% speedup over FUFSA-1.

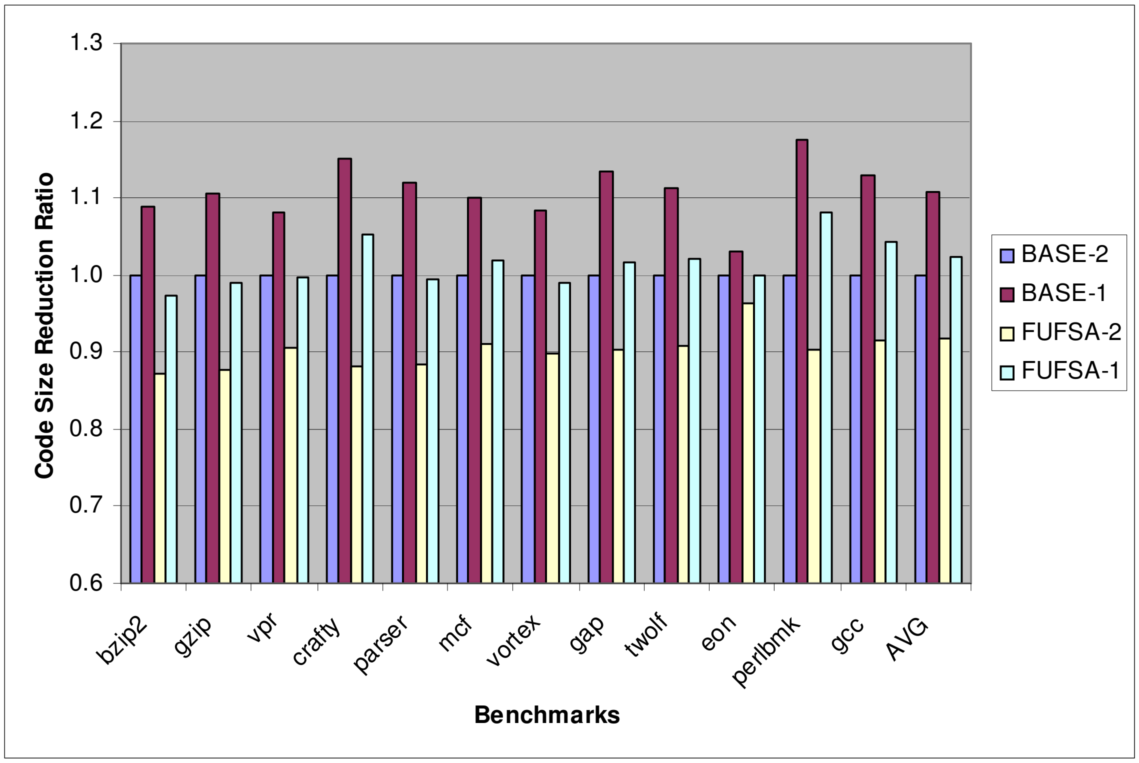


Figure 10: Code size with respect to BASE-2.

Figure 10 compares the code size of the four configurations by measuring the size of the text sections in the generated binaries. Note that a shorter bar in Figure 10 indicates a smaller code size. It is clear that FUFSA-2 is able to generate code that is smaller than BASE-2 does. The static code size from FUFSA-2 is reduced by 9.32% with respect to BASE-2. Furthermore we are able to reduce static code size by about 10% when compressed templates are used to pack instructions, as observed from the reduction achieved by BASE-2 over BASE-1 and FUFSA-2 over FUFSA-1.

To further understand how FUFSA improves performance, we use the PFMON (version 0.06) tool to measure dynamic execution statistics through the performance monitors of Itanium® processors. Due to space limitation, we select two programs, crafty and eon, which benefit the most from the FUFSA approach, and two other programs, bzip2 and perlbmk, which receive only small improvements from the FUFSA approach. Figure 11 shows the distribution of dynamic cycles under the various stall categories for each of the four programs. Readers are referred to [13] for the full description of each stall category. The cycle distributions for each configuration have been normalized with respect to the total cycles of BASE-2.

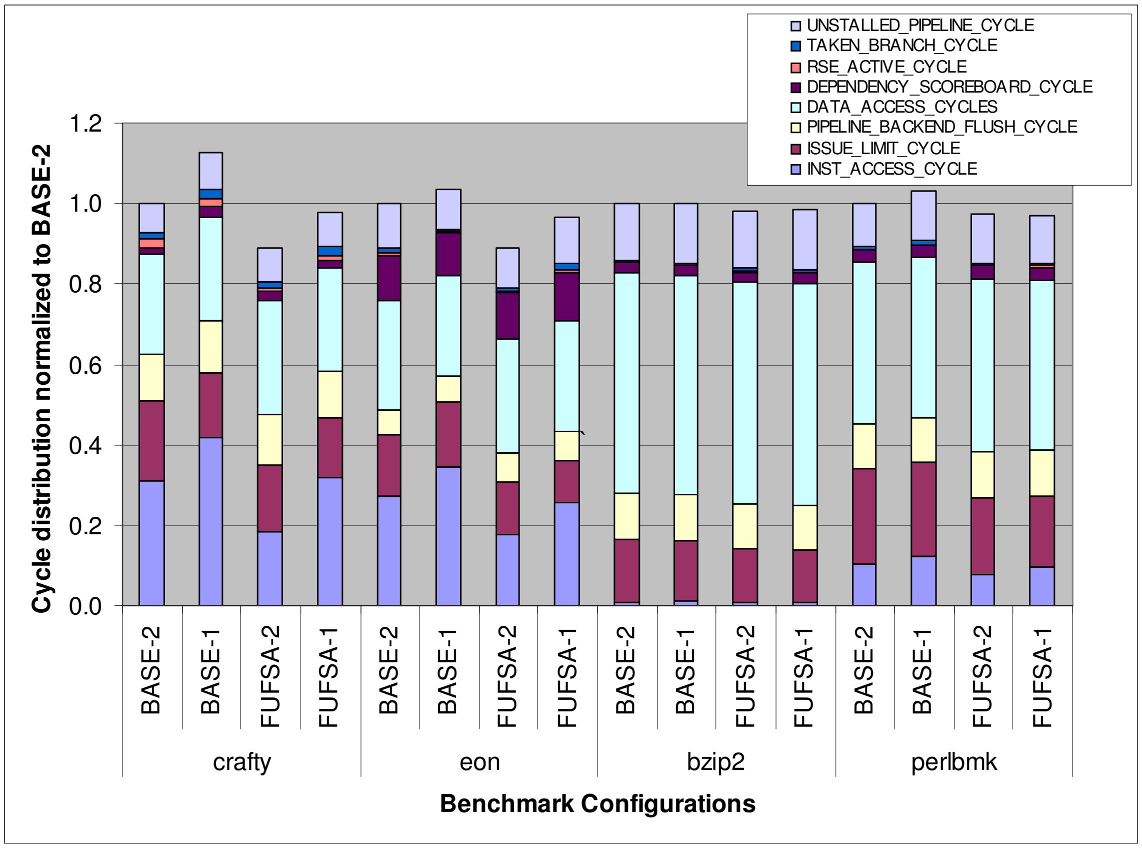


Figure 11: Cycle breakdown for crafty, eon, bzip2 and perlbmk.

As shown in Figure 11, FUFSA mainly reduces cycles in two stall categories – ISSUE_LIMIT_CYCLE and INST_ACCESS_CYCLE. The ISSUE_LIMIT_CYCLE counts all cycle breaks that are due to the explicit insertion of stop bit in the generated code or the implicit insertion of stop bit by the processor when resources are oversubscribed. The BASE configuration does not account for the constraints of instruction templates and dispersal rules during instruction scheduling. Thus it is more aggressive in scheduling instructions into certain cycles even though there is no instruction template to pack them into a single cycle. The independent bundling phase then needs to split these cycles to fit instruction templates. The FUFSA configuration takes into account the effect of instruction templates and dispersal rules during scheduling, avoiding the template selection deficiency. The FUFSA configurations thus have fewer cycles in the ISSUE_LIMIT_CYCLE categories. On the four benchmarks, FUFSA-2 gets 2-4% speedup over BASE-2 and FUFSA-1 obtains 2-6% speedup over BASE-1 due to the reduction of ISSUE_LIMIT_CYCLE.

FUFSA also shows significant cycle reduction over BASE in the INST_ACCESS_CYCLE stall category, which counts cycles lost to I-cache or ITLB misses. FUFSA-2 gains 12% on crafty and 9.5% on eon over BASE-2 in INST_ACCESS_CYCLE. The reduction in I-cache and ITLB misses can be attributed to the fact that the code generated by FUFSA is more compact than the code from BASE. For benchmarks that spend significant execution cycles waiting for I-cache and ITLB misses, such as crafty and eon, FUFSA is able to achieve higher speedup over BASE by reducing the impact from I-cache and ITLB misses. On the other hand, benchmarks that have fewer cycles in the INST_ACCESS_CYCLE category, such as bzip2 and perlbmk, get a lower speedup from FUFSA over BASE.

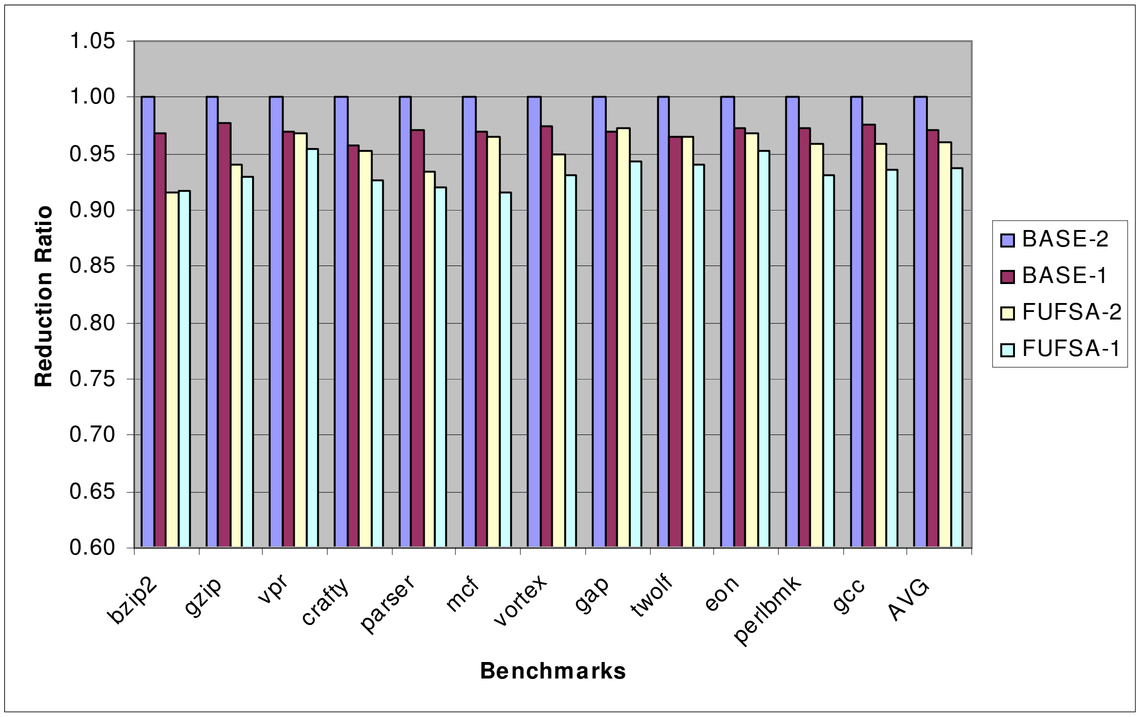


Figure 12: Scheduling time with respect to BASE-2.

We also compare the compilation time spent in instruction scheduling (scheduling time) in Figure 12. We used the cross-build version of ORC hosted on an x86 machine. The compilation time is measured on a workstation with dual 2.4 GHz Pentium IV Xeon processors, 512 KB L2 cache and 512 MB memory. The scheduling time measures the compilation time spent in global scheduling and local scheduling, including the time for the micro-level scheduler and resource modeling. For the BASE configuration, the scheduling time also includes the independent bundling phase for selecting instruction templates. The scheduling time accounts for the majority of the time in the code generator component in the current implementation. The scheduling time at each configuration is normalized to the time at BASE-2. On average, the scheduling time of the FUFSA is about 4% less than the scheduling time of BASE. The scheduling time of using the 1-cycle template selection heuristic is only about 2% less than the scheduling time of using the 2-cycle template selection heuristic. This shows the design of our FSA-based on-the-fly resource management during scheduling provides not only good performance improvements but also compilation time efficiency. It also shows the FUFSA-2 is a better choice in the speedup versus compilation time tradeoff.

6. Related Work

Prior work on modeling hardware resource constraints during scheduling has been mostly on resource (or structural) hazards. Traditional compilers explicitly model the pipeline of the processor by simulating instruction timing. A list of committed resources is maintained in each cycle and tracked by a resource reservation table. Whenever an instruction is considered for scheduling, its resource requirements are checked against the resources already committed in the reservation tables at different cycles. [6, 7, 8, 22] all use the reservation table approach. However, the reservation table approach is less capable of managing instructions that can be handled by multiple types of functional units. Another problem with using resource reservation tables is that the table size is the number of resources times the length of the longest pipeline stage, and every hazard test requires an OR operation on the tables.

Operations in the TriMedia TM1000 mediaprocessor, are issued to different issue slots based their operation types [11]. The mapping from operations to issue slots in TM1000 is fixed. However, the mapping of instructions to FUs in an Itanium® processor is determined from the sequence of instructions at each fetch cycle, based on the instruction templates and dispersal rules. The reservation-table-based approach in [4] for assigning issue slots on TM1000 is not capable of handling the template- and context-sensitive resource constraints of Itanium® processors.

FSA-based approach has the intuitive appeal by modeling a set of valid schedules as a language over the instructions. The model in [5, 19] built FSA directly from the reservation tables. The work in [21] reduced the size of FSA by moving away from using reservation vectors as states. Instead each state encodes all potential structural hazards for all instructions in the pipeline as collision matrix. Additional improvements for the FSA-based approach were proposed in [1] to factor, merge, and reverse automata. Note that because the Itanium® architecture needs a compiler to model resources primarily at hardware issue time, the FSA in our approach takes advantages of that. Reservation-table-based scheduling usually models resources that are needed at issue time and after the issue cycle.

A recent work [15] is an example of the decoupled approach, and it uses an integer linear programming method to model resource constraints as a post-pass local scheduling on assembly code. A subsequent work [25] extends to model certain aspects of global scheduling though still based on the integer linear programming and post-pass approach. Our scheduling approach may involve backtracking to swap functional unit assignments, which is similar to some modulo scheduling work, e.g. [24]. However, the backtracking in our approach is limited within a cycle, whereas the backtracking in modulo scheduling could go much further.

The idea of using a table-driven approach to isolate machine-specific information and retarget a compiler is certainly not new. The work in [9] used a table-driven algorithm to translate a low-level intermediate representation into efficient code sequences for a target machine. Their work focused on selecting sequences of object code instructions for machines with different sets of instructions. In contrast, our work focus on using FSA or a table-driven approach to model resource constraints, in particular under the context of instruction scheduling and ILP-based optimizations. We are more interested in easing the migration to different micro-architectural implementations within the same processor architecture. There is also a similarity between our approach and code selection of CISC-like instructions, where both appear to pack multiple instructions to a wider or more complex instruction sequence. However, the code selection of CISC-like instructions is usually done by looking for particular patterns from a sequence of primitive instructions in the intermediate representation. Although it can also be implemented as a table-driven manner, such code selection is usually not tied to instruction scheduling. In contrast, our approach packs multiple instructions into templates to purposely integrate with instruction scheduling to fully utilize machine resources and latencies. Different combinations of instructions in the templates are also much more flexible than the particular patterns in CISC-like instructions.

Our scheduling approach in this work clearly falls into the line of greedy list scheduling in terms of scheduling length and the overall code size. As mentioned earlier, the high-level scheduling, which drives a cycle-based scheduling, is based on [2] though with a number of enhancements. The greedy nature of the scheduling heuristics selects the candidate with the highest frequency-weighted path length among all ready instructions to schedule into the current cycle. For a set of instructions scheduled into the same cycle, if there exists any legal template combination to fit these instructions, our approach will find at least one such template selection since it will enumerate different combinations of issue slots by attempting to re-assign previously scheduled instructions in the current cycle into some other slots. Since the enumeration space is typically small, the search time has not been an issue. Once the issue slots are fixed, the particular state in the FSA is determined and the first legal template selection also meets the smallest code size within the current cycle since the templates are currently sorted with increasing code sizes. However, our approach does not attempt to achieve the smallest code size for a given region, because the enumeration process for fitting issue slots currently does not look for the smallest code size among all possible template selections with different sets of issue slots. The main reason for choosing our approach is to fit with a practical, production environment, where an optimal solution is usually too expensive to obtain. Greedy list scheduling has been widely adopted, and among other considerations time efficiency has been one important advantage. Hence, we use it as the high-level scheduling in our approach and leverage many known techniques built around it. The way we factor all legal templates under the same FSA state is primary for space efficiency at a small cost of occasionally looking up the list of legal templates.

7. Conclusions

The Intel® Itanium® architecture and its hardware implementations have introduced a new notion of instruction templates and a set of complicated dispersal rules in addition to the traditional pipeline resource hazards. This has stretched the limit of an optimizing compiler, in particular on instruction scheduling, in its ability to model resource constraints effectively and efficiently in the course of generating highly optimized code. In this work, we have extended the FSA-based approach to manage all of the key resource constraints on-the-fly during instruction scheduling. The FSA is built off-line prior to compilation. To largely cut down the number of states in the FSA, each state models the occupied functional units. State transition is triggered by the incoming scheduling candidate, and resource constraints are carefully integrated into a micro-scheduler.

The proposed scheduling approach integrated with resource management has been fully implemented in the Open Research Compiler. The integrated approach shows a clear performance advantage over decoupled approaches with up to 12% speedup and an average of 4.5% improvement across 12 highly optimized SPECint2000 integer programs running on Itanium®-based workstations. This shows the necessity of modeling all resource constraints, including instruction templates and dispersal rules, during scheduling for a high-performing architecture such as the Itanium® architecture. We also demonstrate that the compilation time for our integrated approach is competitive to that of a decoupled approach even with a full modeling of the hardware resources. Furthermore, our machine model and micro-level scheduler are modularized and can be easily retargeted to a newer generation of Itanium® processors.

One possible improvement to our implementation is to encode the supported instruction sequences from the legal template assignments in each state. The encoding would allow a faster check on whether intra-cycle dependences among instructions are supported, eliminating the need to walk through the list of legal template assignments. We would also like to investigate incorporating code size as a first-order consideration into our integrated instruction scheduling and resource management model.

Instruction dispersal rules have become ever more complicated on modern processors for both superscalar and VLIW architectures. This is due to various design consideration, such as performance, power consumption, area, and reconfigurability. Our FSA-based approach on scheduling and resource management is a good framework to model such resource constraints during scheduling beyond the Itanium® architecture. In addition, all VLIW architectures have to pack instructions statically, which can be seen as a form of instruction templates to be modeled by a compiler. We would like to apply our integrated approach to various architectures. One can try to make the set of templates a variable and develop a good schedule for the variable sets of templates, though this remains a challenging problem. We also would like to employ the FU-FSA based approach in a JIT compilation environment where compilation time is critical to the overall performance.

Acknowledgements

We would like to thank members of the ORC teams in Intel and in the Institute of Computing Technology of the Chinese Academy of Sciences for building the ORC compiler, on which our experiments is based. We would also like to thank Kath Knobe for her many helpful comments to improve this paper. Finally we thank the anonymous reviewers for their helpful feedback.

References

[1] V. Bala and N. Rubin, “Efficient Instruction Scheduling Using Finite State Automata,” in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[2] D. Berstein, M. Rodeh, “Global Instruction Scheduling for Superscalar Machines,” in *Proceedings of SIGPLAN’91 Conference on Programming Language Design and Implementation*, pp. 241-255, June 1991.

[3] J. Bharadwaj, K. Menezes, and C. McKinsey, "Wavefront scheduling: path based data representation & scheduling of subgraphs," in *Proeedings. of the 32nd Annual International Symposium on Microarchitecture*, pp. 262-271, 1999.

[4] Z. Chamski, C. Eisenbeis, and E. Rohou, “Flexible Issue Slot Assignment for VLIW Architectures”, INRIA Research Report 3784, October 1999.

[5] E. Davidson, L. Shar, A. Thomas, and J. Patel, “Effective Control for Pipelined Computers,” in *Spring COMPCON-75 digest of papers*. IEEE Computer Society, February 1975.

[6] J. Dehnert and R. Towle, “Compiling for the Cydra-5,” *Journal of Supercomputing*, vol. 7, no. 1-2, pp. 181-227, May 1993.

[7] A. Eichenberger and E. Davidson, "A Reduced Multipipeline Machine Description that Preserves Scheduling Constraints," in *Proceedings of SIGPLAN'96 Conference on Programming Language Design and Implementation*, pp. 12-22, May 1996.

[8] J. Fisher, "Trace scheduling: a technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478-490, July 1981.

[9] S. Glanville and S. Graham, "A New Method for Compiler Code Generation," in *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pp. 231-240, January 1978.

[10] R. Gupta and M. L. Soffa on "Region Scheduling," *IEEE Transactions on Software Engineering*, vol. 16, pp. 421-431, April 1990.

[11] J. Hoogerbrugge and L. Augusteijn, "Instruction Scheduling for TriMedia," *Journal of Instruction-Level Parallelism*, vol. 1, no. 1, February 1999.

[12] Intel, Intel® Itanium® Architecture Software Developer's Manual, Vol. 1, October 2002.

[13] Intel, Intel® Itanium® Processor Reference Manual for Software Optimization, November 2001.

[14] Intel, Itanium® Microarchitecture Knobs API Programmer's Guide, 2001.

[15] D. Kaestner and S. Winkel, "ILP-based Instruction Scheduling for IA-64," in *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 145-154, June 2001.

[16] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg, "The Multiflow Trace Scheduling Compiler," *Journal of Supercomputing*, vol. 7, no. 1-2, pp. 51-142, May 1993.

[17] U. Mahadevan and S. Ramakrishnan "Instruction Scheduling Over Regions: A Framwork for Scheduling Across Basic Blocks," in *Proceedings of the 5th International Conference on Compiler Construction*, Edingburgh, U.K., pp.419-434, April 1994.

[18] S. Moon and K. Ebcioglu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 55-71, December 1992.

[19] T. Muller, "Employing Finite Automata for Resource Scheduling," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

[20] Open Research Compiler (ORC) 2.0, http://ipf-orc.sourceforge.net, January 2003.

[21] T. Proebsting and C. Fraser, "Detecting Pipeline Structural Hazards Quickly," in *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 280-286, January 1994.

[22] B. Rau, M. Schlansker, and P. Tirumalai, "Code Generation Schemes for Modulo Scheduled Loops," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[23] B. Rau and J. Fisher, "Instruction-level Parallel Processing: History, Overview, and Perspective," *Journal of Supercomputing*, vol. 7, no. 1-2, pp. 9-50, May 1993.

[24] B. Rau, "Iterative Modulo Scheduling," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, December 1994.

[25] S. Winkel, "Optimal Global Scheduling for Itanium® Processor Family," in *Proceeding. of the 2nd EPIC Compiler and Architecture Workshop* (EPIC-2), November 2002.