Efficient Resource Management during Instruction Scheduling for the EPIC Architecture

Dong-Yuan Chen*, Lixia Liu*, Chen Fu**, Shuxin Yang**,
Chengyong Wu**, Roy Ju*

*Intel Labs
Intel Corporation
Santa Clara, CA 95052

**Institute of Computing Technology
Chinese Academy of Sciences
Beijing, P. R. China

ABSTRACT

Effective modeling and management of hardware resources have always been critical toward generating highly efficient code in static compilers. With Just-In-Time compilation and dynamic optimization, the efficiency of resource modeling and management has become equally important. The introduction of instruction templates and instruction dispersal rules on the EPIC architecture, in addition to the traditional pipeline resource hazards, have stretched the limit of an optimizing compiler, in particular on instruction scheduling, in its ability to manage resource constraints effectively and efficiently. In this work, we have extended a finite state automaton (FSA)-based approach to manage all key resource constraints of an EPIC architecture on-the-fly during instruction scheduling. These intertwined constraints are carefully factored into the FSA states to achieve both time and space efficiency. We have fully integrated the FSA-based resource management into the instruction scheduler in the Open Research Compiler for EPIC 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 machine when compares to an instruction scheduler with decoupled resource management. In the mean time, the instruction scheduling time of our approach is reduced by 4% on average.

1. INTRODUCTION

The Explicitly Parallel Instruction Computing (EPIC) architecture exemplified by the Itanium Processor Family (IPF) requires 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 EPIC architecture [10] introduces the notion of instruction bundles and templates, which limit the instruction mixes presented to the hardware so that these instructions can be dispatched efficiently to proper execution units. Instruction templates present a further challenge for resource management by imposing new constraints on instruction packing and dispatching. In fact, such new constraints are not unique to EPIC. Modern processors tend to segment the execution units across different types (e.g. memory, ALU, floating-point, branch, etc.) and within the same types. A given instruction can sometimes be issued only to a particular unit within the same type, and such decision could even be affected by surrounding instructions.

When dealing with these new and complex resource constraints, a simple but sub-optimal solution is to perform a separate instruction packing phase according to templates and instruction dispatching constraints 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 EPIC, which can issue up to three instructions per cycle (bundled by a template). 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.

```
ld  a = [x]                    {mii:   ld  a = [x]
add b = y, e                           add b = y, e
ld  y = [f] ;; // can't fit            add x = h, i ;;}
ld  c = [g]                    {mmi:   ld  y = [f]
add x = h, I                           ld  c = [g]
add d = j, k ;;                        add d = j, k ;;}
```

(a) Decoupled scheduling and template selection.

(b) Integrated scheduling and template selection.

Figure 1. An example to contrast decoupled vs. integrated approaches.

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 phase to pack instructions into templates 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. 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 an instruction scheduling phase is integrated with full resource management including template selection, it can achieve the optimal scheduling using two templates (MII and MMI) in two respective cycles as shown in Figure 1(b).

Instruction scheduling has already been one of the most complex phases in an optimizing compiler. Extension to model different hardware resources, including instruction templates, is a challenging task given the large search space for a valid and profitable combination. This task is further complicated by a number of additional practical issues, such as the instruction dispersal rules for dispatching an instruction sequence onto execution units, the compressed templates for packing instructions from different cycles, the one-to-many mapping possibility from instructions to functional units, etc.

In this work, we propose to use finite-state automata (FSA) to model all of these resource constraints under instruction scheduling. Each state represents the currently occupied functional units of a cycle and is augmented with instruction templates and dispatching information. The transition of states is triggered by the incoming scheduling candidate instruction. Our key contribution is to extend an FSA approach to successfully model the new notion of instruction template and a set of complicated dispersal rules on the EPIC architecture in addition to the traditional pipeline hazards. Our experimental results, based on the implementation in ORC, show that modeling these additional resource constraints is crucial to achieve high performance on Itanium. The modeling of these different but correlated constraints is done with both time and space efficiency.

We have also encapsulated the modeling and management of hardware resources into a modularized machine model to allow easy migration to future implementations of the EPIC architecture or any future processors with similar resource constraints.

In the rest of the paper, Section 2 provides background information and definitions of terminology. Section 3 details the concept of functional-unit based finite-state automata and its construction prior to compilation. Section 4 discusses how to perform instruction scheduling by taking into account different resource constraints based on the FSA. Section 5 shows the experimental results to compare our integrated approach with decoupled approaches. Section 6 discusses the related work, and Section 7 concludes this paper.

2. BACKGROUND

2.1 EPIC and Terminology

EPIC [10] 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* dictates that the instructions before and after the stop bit 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. We call the two *compressed templates* because they allow the packing of instructions from different cycles into smaller code size. On EPIC, flow and output register dependences are generally disallowed (with a few exceptions) within a cycle, but register anti-dependences are generally allowed.

Each implementation of the EPIC architecture has its own micro-architectural features. For example, the Itanium processor [11] can issue and execute up to six instructions (two bundles) per cycle. There are 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 are used to model specific pipelined execution units. 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 also has different latencies among dependent instructions and bypasses between different execution pipelines, resulting in varying latencies between the same pair of dependent instructions when they are dispatched to different FUs. Each processor generation has a unique set of *instruction dispersal rules* that describes how each instruction is dispersed to a FU in an

instruction sequence. As an example, 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. For example, in an MII bundle, the first I instruction will be issued to the I0 unit regardless whether it can execute on I1 or not. If the second I instruction can be executed only on I0, which is already taken, it will be forced into the next cycle. A detailed description of all of these micro-architectural features for the Itanium processor can be found in [11].

2.2 High-Level 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. The high-level instruction scheduling is in charge of determining the issue cycle of each instruction according to dependences and instruction execution latencies. It applies ILP (instruction-level parallelism) enabling transformations such as speculation, sets the scheduling priority of instructions, and decides when an instruction is to be executed. The micro-level scheduling takes care of the placement of instructions and resource management within an execution cycle, based on the intra-cycle instruction dependences specified by the high-level scheduler. The micro-level scheduling shields the complicated resource management of the EPIC architecture from high-level scheduling algorithm, making migrating to a different generation of EPIC processors easier.

Both the high-level and micro-level instruction schedulers retrieve machine-specific information from the machine description structures prepared off-line by a machine model builder (MM builder). The machine description structures specify the numbers and types of machine resources (such as machine width, functional units, and templates), the latency of instructions, the pipeline bypass constraints, etc. The functional-unit based FSA is also part of machine description

structures built off-line. The machine description structures become part of the static data in the compiler executable after the compiler is built. Instead of defining a new machine description language [9], the MM builder constructs the machine description from a published Itanium microarchitecture parameter file [12] that describes the architecture and micro-architecture details of an EPIC processor.

3. 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, 19]. 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 is not difficult to model, the instruction templates and the instruction dispersal rules that map instructions to their respective functional units introduce new challenges. The latency from the source instruction to the destination instruction depends not only on the type of source instruction but also on the functional unit where it is executed. The functional unit assigned to execute an instruction is determined by its slot position in a bundle and the template of the bundle. Hence selecting the instruction template and placing instructions into the proper slot positions in a bundle is critical in 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 bundle slot for the instruction. The instruction latency and the executing functional unit of the instruction are then derived from the instruction template and bundle slot assigned per instruction dispersal rules. All possible template assignments are enumerated dynamically at each scheduling attempt to select the slot and template, but this approach results in significant compile-time overhead.

One could improve the dynamic enumeration of template assignments by building a template-based FSA off-line and using the FSA to guide the selection of template assignments. The template-based FSA models the template assignment and the slot usage of an execution cycle. Each state in a template-based FSA comprises of all the possible selections of instruction templates under certain slot usage in a single execution cycle. When an instruction is scheduled, an unused slot *S* is picked for the instruction and the FSA is transited to the next state where the slot *S* becomes taken. Template assignment is then selected from the legal template assignment for the state. Size is one major problem in a template-based FSA approach. For the Itanium processor that can issue up to two bundles per cycle, there are at least 68 possible template assignments in a cycle, after eliminating invalid template assignments that may oversubscribe available 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) that models the resource usage of a schedule cycle. 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 those states with at least one legal template assignment are included in the FU-FSA. The list of legal template assignments for each state is built off-line by the MM builder.

When an instruction is being scheduled, the micro-level scheduler looks for a FU *P* for the instruction that can lead to a legal transition in the FU-FSA. A state transition is legal if the new

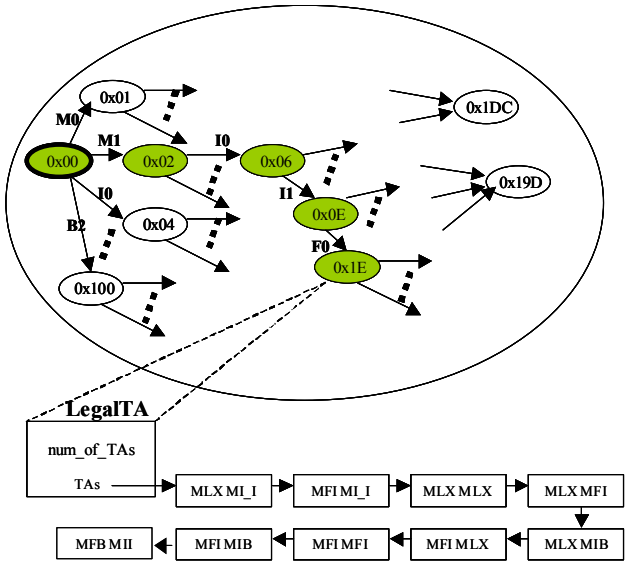

Figure 2. Functional unit based finite state automata.

```
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].grps;
      }
    }
    IF (PV is in FSA) {
      Sort FSA[PV].grps according to
        priority criteria;
    }
  }
  Build FSA transition table;
}
```

Figure 3. Pseudo code for building FU-based FSA.

FU usage set is a valid FU-FSA state and at least one template assignment for the new state satisfies the dependence constraints for all instructions currently scheduled in the cycle per the template constraints and instruction dispersal rules. When there is no intra-cycle instruction dependence, the legal FU-FSA transition check is simply verifying the new FU usage set is indeed a state in the FU-FSA. 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 legal FU-FSA transition check 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 cases, the final selection of template assignment is needed only when scheduling for a cycle has completed, instead of done at every scheduling attempt.

The FU-FSA is much more space efficient than the template-based FSA. The Itanium processor has 9 FUs (2 M's, 2 I's, 2 F's and 3 B's,) therefore the FU-FSA has at most 2^9 states, much smaller than that of a template-based FSA. After eliminating illegal states, the FU-FSA for Itanium 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.

Figure 2 illustrates a FU-FSA where the FU usage set in each state is represented as a bit vector. When a FU is assigned to an instruction during instruction scheduling, the current state transits to a new state by setting the bit corresponding to the newly occupied FU. For example, if the current state is 0x00 and the new instruction occupies the M0 unit (bit 0 in the bit vector), the new state will be 0x01. Each state has a structure *LegalTA*, which contains a list of template assignments that have been pre-determined to be legal for that state per instruction dispersal rules. Figure 2 also highlights a possible sequence of transitions of the FU-FSA from the initial state 0x00 to the state 0x1E, following the transition path M1->I0->I1->F0.

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, we can optimize the template selection for minimal code size (or bundle count) as well as for other objectives.

For instance, we arrange the list of template assignments according to the following priorities:

1. Smaller bundle count in a template assignment.
2. Template assignments with compressed template(s).

These heuristics reduce the code size of the scheduled instructions. A smaller code size in general leads to better performance due to a reduction in I-cache misses. Other heuristics can be applied. For example, it may be desirable to place an MFB template before an MFI template

since the MFB template can facilitate the introduction of branch prediction instructions in a later phase.

The pseudo code *BuildFSA* in Figure 3 outlines the algorithm that constructs the FU-FSA for the Itanium processor. It can be easily extended to generate a FU-FSA for any EPIC 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 for the state *PV*, that is, *FSA[PV]*. After all template assignments for the FU usage set *PV* are examined, the list of legal template assignments for *PV* is sorted according to the priority ordering described above. Finally the FU-FSA transition table is constructed after all legal states of FU-FSA are included.

4. INSTRUCTION SCHEDULER WITH INTEGRATED RESOURCE MANAGEMENT

With the help of a FU-FSA based micro-level scheduler, the instruction scheduler can focus on high-level scheduling decision while still models detail low-level resources efficiently. The flow chart in Figure 4 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.

The instruction scheduler repeatedly picks the best candidate from a list of instructions that are ready for scheduling. It then consults the micro-level scheduler through the *IssueOp* function to check for resource availability. For Itanium processor, the *IssueOp* function must determine

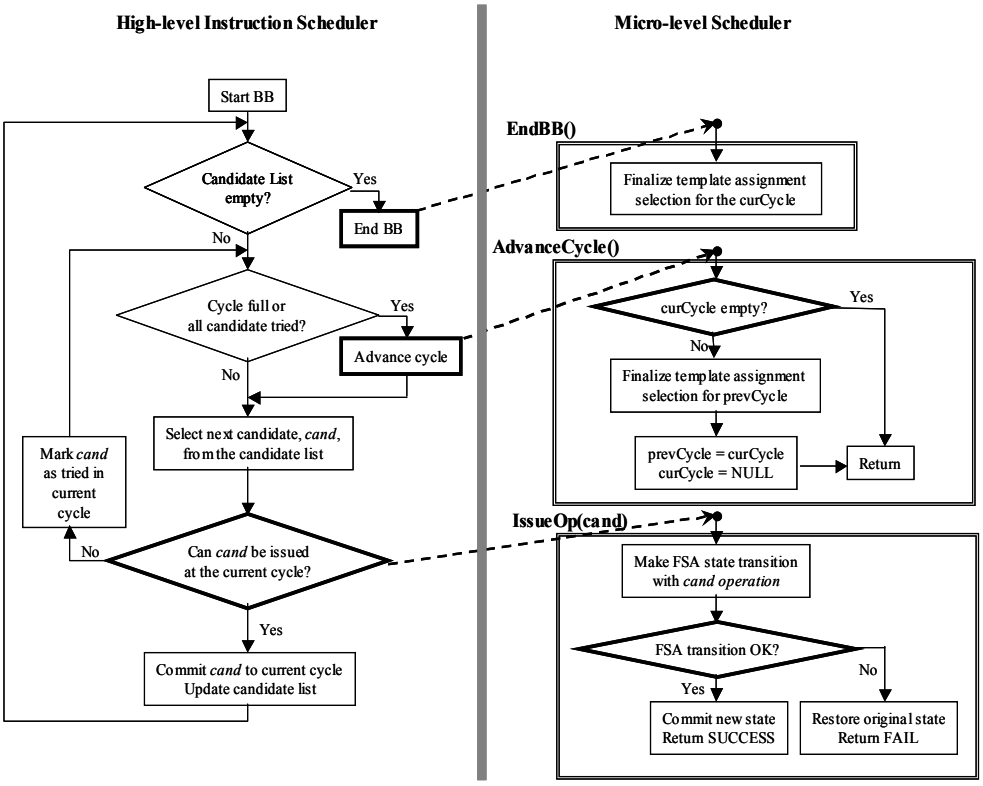

Figure 4. Interaction between high-level and micro-level instruction scheduling.

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. In our FU-FSA approach, these tasks are accomplished by simply picking an available FU for the candidate instruction and then consulting the FU-FSA for a legal state transition.

If *IssueOp* completes successfully, the candidate instruction is committed to the cycle and the instruction scheduler moves on to schedule the next candidate instruction. If *IssueOp* reports that it is unable to fit the instruction in the current cycle, the instruction scheduler marks the candidate instruction as being tried already. It then picks another candidate instruction from the candidate list to schedule in the current cycle. Once the current cycle is full or has no more candidate instruction to try, the current cycle is closed and the instruction scheduler advances to schedule for the next cycle.

```
MakeFSAStateTransition(op, cycle)
{
  func_unit = all FUs that op can be issued to;
  free_unit = all unoccupied FUs in cycle;

  // Try available FUs.
  candidate_unit = func_unit & free_unit;
  FOREACH FU in candidate_unit DO {
    Record op issued to FU in cycle;
    state = cycle->FU_state; // new state
    IF (FSA[state] is valid) {
      IF (intra-cycle dependence in cycle) {
        FOREACH ta in FSA[state].grps DO {
          IF (ChkCycleDep(cycle, ta) == TRUE)
             RETURN TRUE;     // success
        }
      } ELSE RETURN TRUE;
    }
    Back out op from FU in cycle;
  }

  // Try permuting FU assignments.
  candidate_unit = func_unit & ~free_unit;
  FOREACH FU in candidate_unit 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 (MakeFSAStateTransition(old_op, cycle)
        == TRUE)
      RETURN TRUE;
    Back out op from FU in cycle;
    Record old_op issued to FU in cycle;
  }
  RETURN FALSE;
}
```

Figure 5. Pseudo code for IssueOp and making FSA state transition.

There are several approaches on when to finalize the template assignment for each cycle. One could choose 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 thus 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. Deferring the template assignment and doing it for several cycles at once has the advantage of achieving a more compact code size by exploiting compressed templates to pact instructions in adjacent cycles. However, in addition to extra compilation time, larger space is needed to keep the scheduling states of several cycles around.

Our instruction scheduler employs a 2-cycle template selection heuristic that takes a middle ground to obtain the key advantages of both approaches, but gears toward finalizing template

assignment on-the-fly in a cycle-by-cycle fashion. 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 three schedule-cycle buffers need to be maintained during instruction scheduling, namely, a previous cycle (*prevCycle*), the current cycle (*curCycle*), and a scratch cycle buffer (*tempCycle*). 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. Such an approach allows a better utilization of compressed templates to connect the previous cycle and the current cycle with a minimum space overhead. We consider this approach to be a good balance between the quality of generated code and the space and time efficiency of the instruction scheduler.

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

The pseudo code in Figure 5 illustrates how the FU-FSA is used to check for legal FSA transition as well as template assignment constraints and instruction dependence. All transient states before a FU for the new instruction is finalized are maintained in the scratch schedule-cycle buffer, *tempCycle*. The function *IssueOp* relies on the function *MakeFSAStateTransition* to perform all the checks necessary to make a state transition. Inside the *MakeFSAStateTransition* function, unoccupied FUs for the new instruction, *inst*, are selected first, as shown in the first FOREACH loop over the candidate FUs. When a tentative FU is selected for *inst*, *tempCycle* is updated to reflect the assignment. The new FU usage set (*cycle->FU_state*) is checked with the FU-FSA to make sure this is a legal transition. Furthermore, if there are dependences among instructions within the cycle, the list of legal template assignments 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 assignment.

If *inst* fails to use any of the unoccupied FUs, occupied but valid FUs for *inst* are tried next. This is intended to 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 in *MakeFSAStateTransition* performs the necessary FU re-mapping. A FU locking mechanism is in place to avoid 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 FU-FSA Based Resource Management and Software Pipelining

The FU-FSA model can be integrated into a scalar instruction scheduler as well as a software-pipelining scheduler. The FU-FSA-based micro-level scheduler works on cycle-level resource management. It relies on the high-level scheduler to provide intra-cycle dependence information among instructions. For illustration purpose, let us assume that the high-level scheduler for software pipelining is a modulo scheduler. The modulo scheduler can model the modulo resource reservation table as N consecutive scheduling cycles in the FU-FSA-based micro-scheduler, where N is the initiation interval of the pipelined schedule under construction. When querying micro-level scheduler for resource availability at a schedule cycle C, the modulo scheduler will send the query to the schedule cycle (C mod N) in the micro-level scheduler. The modulo scheduler also provides the micro-level scheduler information on intra-modulo-cycle instruction dependences that account for both loop-independent and loop-carried dependences. The final template assignments for all the N schedule cycles will be considered and selected at once after a final schedule is constructed by the modulo scheduler.

5. EXPERIMENTAL RESULTS

The proposed instruction scheduling integrated with FU-FSA-based resource management for the EPIC architecture has been fully implemented in the EPIC Open Research Compiler (ORC) [18]. We compare our integrated approach with the approaches of decoupled instruction scheduling and resource management 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 scheduling [2,3,7,8,14,15,16,21]. 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, which containing multiple basic blocks with internal control flow transfers. The enhanced cost function is based on the path lengths (to the last cycle) weighted by the execution frequency in a global dependence DAG built for the scheduling region. The global instruction scheduling also utilizes special EPIC architectural features and performs control and data speculation to move load instructions across branches and aliasing. 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, pipeline latency, issue width, and the functional

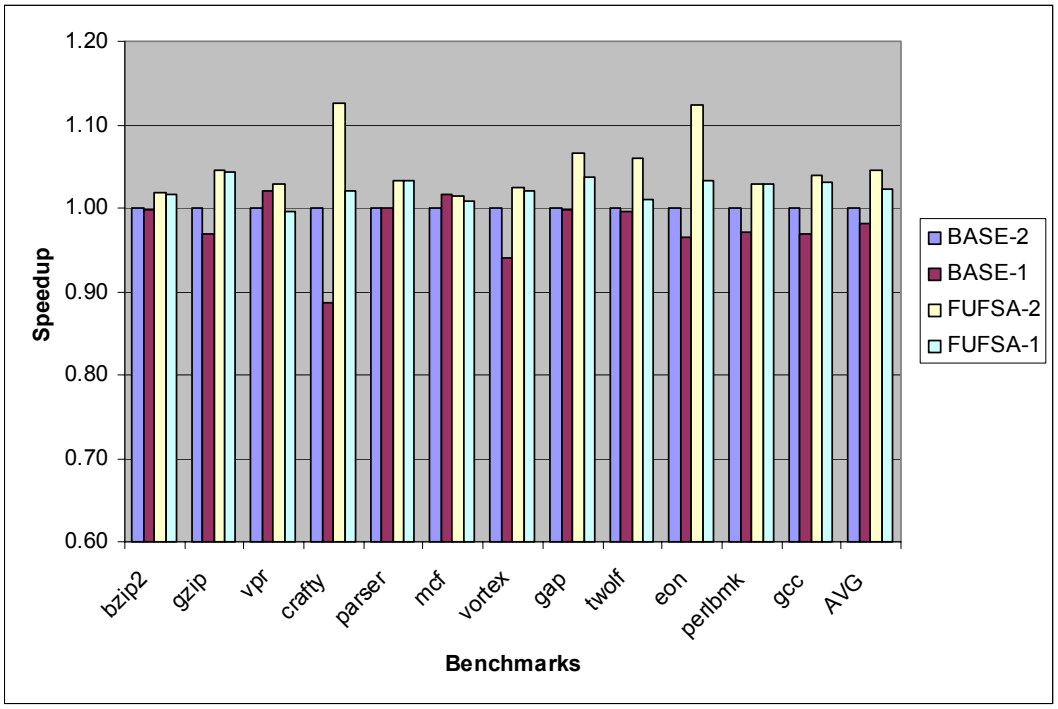

Figure 6. CPU cycles speedup over BASE-2.

unit availability. But instruction dispersal rules and templates are only taken into account during a subsequent, independent bundling phase after the local scheduling that is dedicated to pack instructions under templates. The subsequent bundling phase uses the same FU-FSA-based machine model and micro-level scheduler in much the same way as in FUFSA except that it cannot 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 pipeline latency, issue width, 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 disables the use of two compressed templates, MI_I and M_MI. These configurations are called BASE-1 and FUFSA-1 respectively.

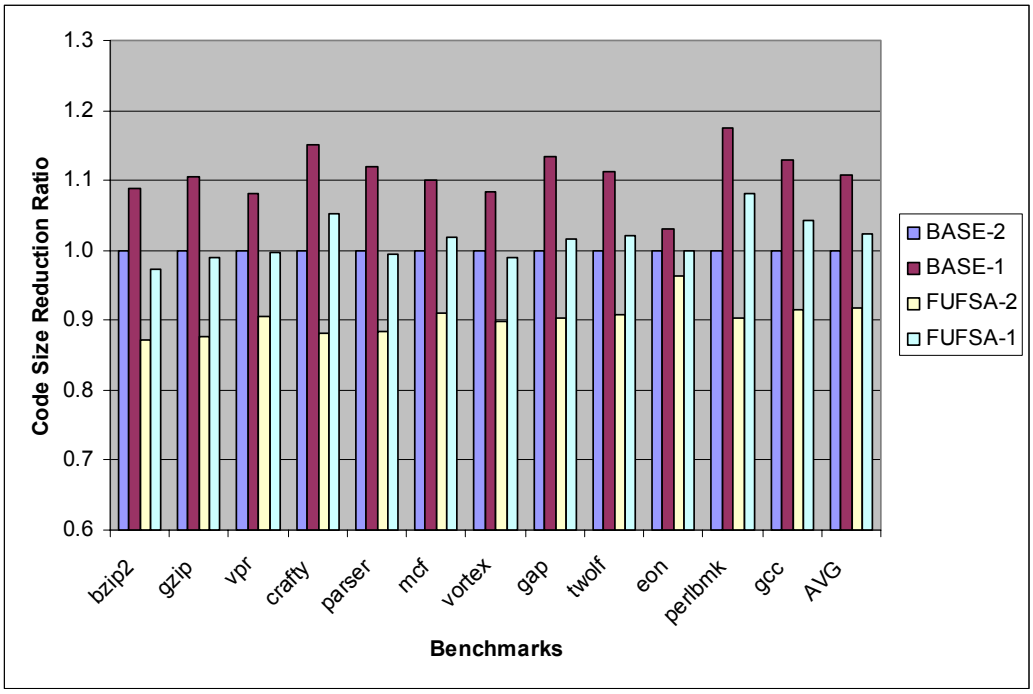

Figure 7. Code Size with respect to BASE-2.

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-specific optimizations. The generated codes are run and measured on a 733 MHz Itanium Workstation with 2 MB L3 cache and 1 GB memory, running RedHat 7.1 version of Linux.

Figure 6 shows the speedup in CPU cycles off 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

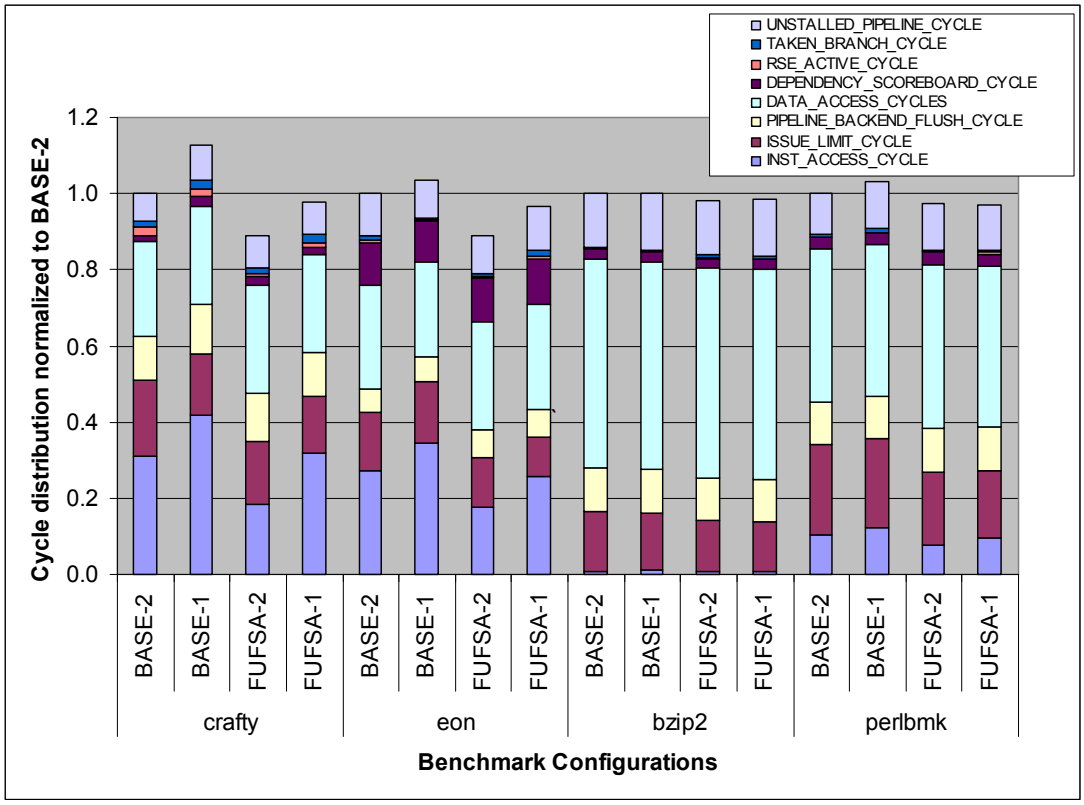

Figure 8. Cycle breakdown for crafty, eon, bzip2 and perlbmk.

selection heuristic performs better than the 1-cycle template selection heuristic when packing instructions into compressed templates. On average, BASE-2 get a 1.8% speedup over BASE-1 and FUFSA-2 obtains a 2.26% speedup over FUFSA-1.

Figure 7 compares the code size of the four configurations by measuring the text section sizes of the generated binaries. Note that the shorter the bars are, the smaller code sizes are. It is clear that FUFSA-2 is able to generate code that is much more compact than the code generated by BASE-2. 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 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 Itanium performance monitors. 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 8 shows the dynamic cycles distributed into the Itanium stall categories for each of the four programs. Readers are referred to [11] for the exact description of stall each category. The cycle distributions for each configuration have been normalized with respect to the total cycles of BASE-2.

As shown in Figure 8, 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 the instruction templates. The FUFSA configuration takes into account the effect of instruction templates and dispersal rules, avoiding the template selection problem. 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 obtained can be attributed to the fact that the code generated by FUFSA is more compact than the code from BASE. For benchmarks that have significant execution time waiting for I-

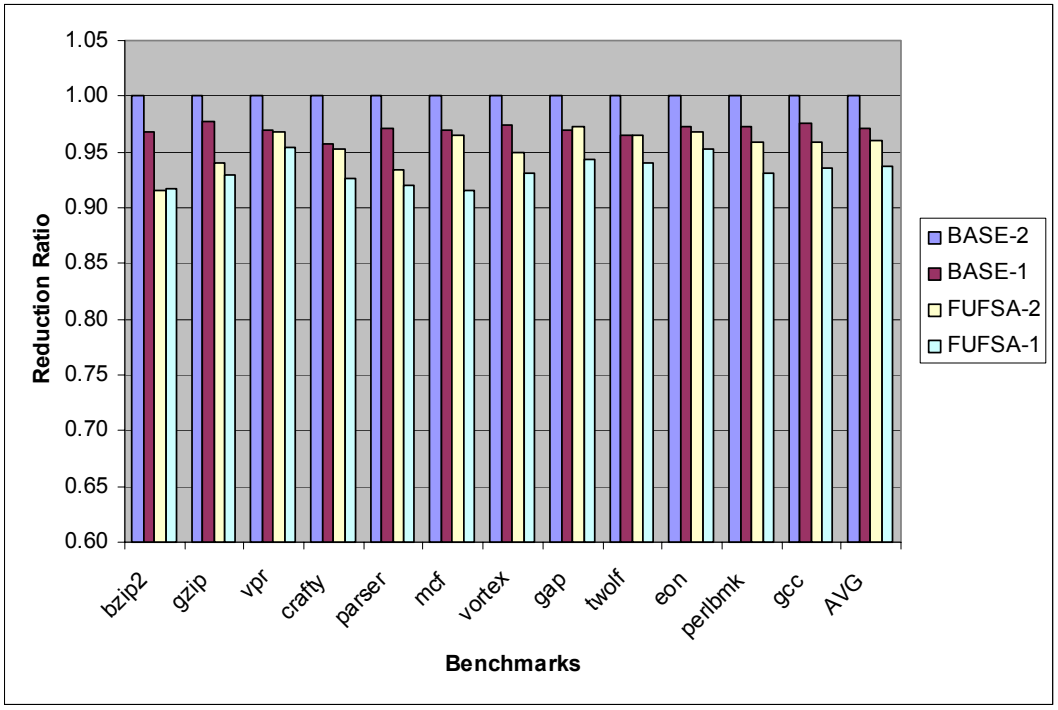

Figure 9. Scheduling time with respect to BASE-2.

cache and ITLB misses, e.g., crafty and eon, FUFSA is able to achieve much higher speedup over BASE by reducing the impact from I-cache and ITLB misses. On the other hand, benchmarks, such as bzip2 and perlbmk, have much fewer cycles in the INST_ACCESS_CYCLE category. These benchmarks thus have a lower speedup from FUFSA over BASE.

We also compare the time spent in instruction scheduling (scheduling time). 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[1]. 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

[1] ORC does provide the capability of native build on an Itanium machine. However, since x86 machines are very popular resources, most of the ORC development environments remain as cross-build.

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. And 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 verse compilation time tradeoff.

6. RELATED WORK

Prior works on modeling hardware resource constraints during scheduling have 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. The works in [5, 6, 7, 20] have been using the reservation table approach. However, this 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. FSA-based approach has the intuitive appeal by modeling a set of valid schedules as a language over the instructions. The model in [4, 17] built FSA directly from the reservation tables. The work in [19] 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.

A recent work [13] 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 [23] 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 issue ports, which is similar to some modulo scheduling work, e.g. [22]. However, the backtracking in our approach is limited within a cycle, whereas the backtracking in modulo scheduling could go much further.

7. CONCLUSIONS

The EPIC 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 CPU2K integer programs running on Itanium

machines. Attempts to extend a traditional scheduler to model part of the resource constraints may sound appealing with an ease of implementation but fail to provide viable performance in our experiments. This shows the necessity of modeling the resource constraints during scheduling for a high-performing architecture such as EPIC. We also demonstrate that the compilation time for our integrated approach is competitive to the compilation time of decoupled approaches 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 EPIC hardware.

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 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 EPIC. In addition, all VLIW architectures have one way or another to bundle 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. 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.

REFERENCES

[1] V. Bala and N. Rubin, “Efficient Instruction Scheduling Using Finite State Automata,” *Proc. of the 28th Annual International Symposium on Microarchitecture*, pp. 46-56, Nov. 1995.

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

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

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

[5] J. Dehnert and R. Towle, “Compiling for the Cydra-5,” *Journal of Supercomputing*, 7:181-227, May 1993.

[6] A. Eichenberger and E. Davidson, “A Reduced Multipipeline Machine Description that Preserves Scheduling Constraints,” *Proc. of SIGPLAN ’96 Conference on Programming Language Design and Implementation*, pp. 12-22, May 1996.

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

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

[9] J. Gyllenhaal, W. Hwu, and B. Rau, “Optimization of Machine Descriptions for Efficient Use,” *Proc. of the 29th Annual International Symposium on Microarchitecture*, pp. 349-358, Dec. 1996.

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

[11] Intel, *Intel Itanium Processor Reference Manual for Software Optimization*, November 2001.

[12] Intel, *Itanium Microarchitecture Knobs API Programmer's Guide*, 2001.

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

[14] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O’Donnell, and J. Ruttenberg, “The Multiflow Trace Scheduling Compiler,” *Journal of Supercomputing*, 7(1,2):51-142, March 1993.

[15] U. Mahadevan and S. Ramakrishnan "Instruction Scheduling Over Regions: A Framwork for scheduling Across Basic Blocks" *Proc. of 5th Int. Conf. on Compiler Construction*, Edingburgh, U.K., pp.419 - 434, Apr. 1994.

[16] S. Moon and K. Ebcioglu, “An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors,” *Proc. of the 25th Annual International Symposium on Microarchitecture*, pp. 55-71, Dec. 1992.

[17] T. Muller, “Employing Finite Automata for Resource Scheduling,” *Proc. of the 26th Annual International Symposium on Microarchitecture*, Dec. 1993.

[18] Open Research Compiler (ORC), http://ipf-orc.sourceforge.net, Jan. 2002.

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

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

[21] B. Rau and J. Fisher, "Instruction-level Parallel Processing: History, Overview, and Perspective," *Journal of Supercomputing*, 7:9-50, May 1993.

[22] B. Rau, "Iterative Modulo Scheduling," *Proc. of the 27th Annual International Symposium on Microarchitecture*, Dec. 1994.

[23] S. Winkel, "Optimal Global Scheduling for Itanium Processor Family," *Proc. of the 2nd EPIC Compiler and Architecture Workshop (EPIC-2)*, Nov. 2002.