Properties of Rescheduling Size Invariance for Dynamic Rescheduling-Based VLIW Cross-Generation Compatibility

Thomas M. Conte
Sumedh W. Sathaye

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695–7911
(919) 515-7983
{conte,swsathay}@eos.ncsu.edu

Abstract

The object-code compatibility problem in VLIW architectures stems from their statically scheduled nature. *Dynamic rescheduling (DR)* [1] is a technique to solve the compatibility problem in VLIWs. DR reschedules program code pages at *first-time page faults* i.e., when the code pages are accessed for the first time during execution. Treating a page of code as the unit of rescheduling makes it susceptible to hazard of changes in the page-size during the process of rescheduling. This paper proves that the changes in the page-size are only due to insertion and/or deletion of NOPs in the code. Further, it presents an ISA encoding called *list encoding*, which does not require explicit encoding of the NOPs in the code. A property of the encoding called *rescheduling-size invariance (RSI)* is presented and it is proved that the list encoding satisfies this property.

1 Introduction

The object-code compatibility problem in VLIW architectures stems from their statically scheduled nature. The compiler for a VLIW machine schedules code for a specific machine model (or a machine generation), for precise, cycle-by-cycle execution at run-time. The machine model assumptions for a given code schedule are unique, and so are its semantics. Thus, code scheduled for one VLIW is not guaranteed to execute correctly on a different VLIW model. This is a characteristic of VLIWs often cited as an impediment to VLIWs becoming a general-purpose computing paradigm [2]. An example to illustrate this is shown in Figures 1, 2, and 3. Figure 1 shows an example VLIW schedule for a machine model which has two IALUs, one Load unit, one Multiply unit, and one Store unit. Execution latencies

of these units are as indicated. Let this machine generation be known as generation X. Figure 2 shows the next-generation (generation $X+1$) VLIW where the Multiply and Load

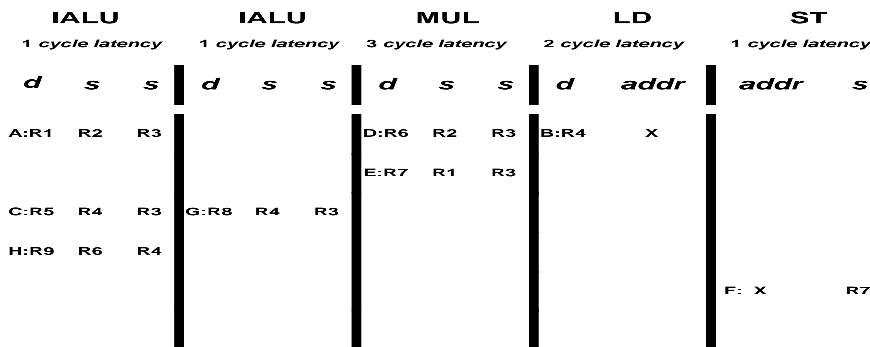

Figure 1: Scheduled code for VLIW Generation X.

latencies have changed to 4 and 3 cycles respectively. The generation X schedule will not execute correctly on this machine due to the flow dependence between operations B and C, between D and H, and between E and F. Figure 3 shows the schedule for a generation $X+n$ machine which includes an additional multiplier. The latencies of all FU*s* remain as shown in Figure 1. Code scheduled for this new machine will not execute correctly on the older machines because the code has been moved in order to take advantage of the additional multiplier. (In particular, E and F have been moved.) There is no trivial way to adapt this schedule to the older machines. This is the case of downward incompatibility between generations. In this situation, if different generations of machines share binaries (e.g., via a file server), compatibility requires either a mechanism to adjust the schedule or a different set of binaries for each generation. One way to avoid the compatibility problem would be to maintain binary executables customized to run on each new VLIW generation. But this would not only violate the copy-protection rules, but would also increase the disk-space usage. Alternatively, program executables may be translated or *rescheduled* for the target machine model to achieve compatibility. This can be done in hardware or in software. The hardware approach adds superscalar-style run-time scheduling hardware to a VLIW [3], [4], [5], [6], [7]. The principle disadvantage of this approach is that it adds to the complexity of the hardware and may potentially stretch cycle time of the machine if the rescheduling hardware falls in the critical path. The software approach is to perform off-line compilation and scheduling of the program from the source code or from decorated object modules (`.o` files). Code rescheduled in this manner yields better relative speedups, but the technique is cumbersome to use due to its off-line nature. It could also imply violation of copy protection. *Dynamic Rescheduling (DR)* [1], is a third alternative to solve the compatibility problem. Under dynamic rescheduling, a program binary compiled for a given VLIW generation machine model is allowed to run on a different VLIW generation. At each *first-time page fault* (a page-fault that occurs when a code page is accessed for the first time during program

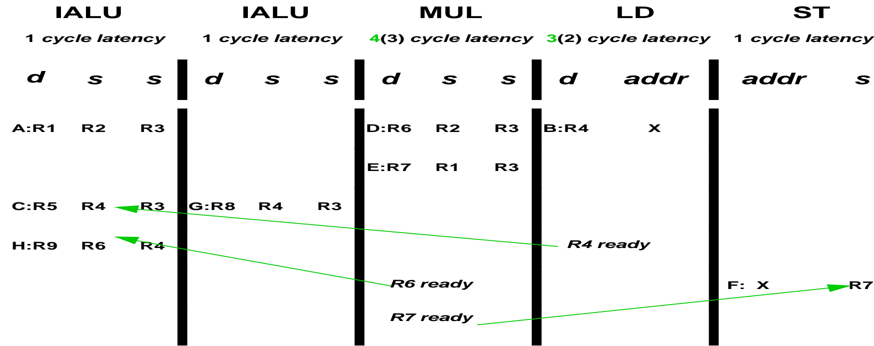

Figure 2: Generation $X + 1$ VLIW schedule: Incompatibility due to changes in functional unit latencies (shown by arrows). The old latencies are shown in parentheses. Operations C, H, and, F now produce incorrect results because of the new latencies for operations B, D, and E.

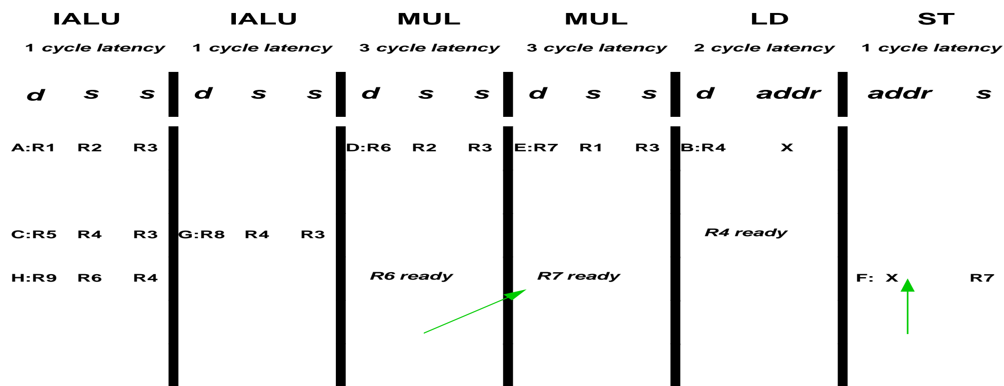

Figure 3: Generation $X + n$ schedule: downward incompatibility due to change in the VLIW machine organization. No trivial way to translate new schedule to older generation.

execution), the page fault handler invokes a module called the rescheduler, to reschedule the page for that host. Rescheduled code pages are cached in a special area of the file system for future use to avoid repeated translations.

Since the dynamic rescheduling technique translates the code on a per-page basis, it is susceptible to the hazard of changes in the page-size due to the process of rescheduling. If the changes in the machine model across the generation warrant addition and/or deletion of NOPs to/from the page, it would lead to page overflow or an underflow. This paper discusses a technique called *list encoding* for the ISA and proves the property of *rescheduling-size invariance (RSI)*, which guarantees that there is no code-size change due to dynamic rescheduling. The organization of this paper is as follows. Section 2 presents the terminology used in this paper. Section 3 briefly describes dynamic rescheduling and demonstrates the problem of code-size change with an example. Section 4 introduces the concept of rescheduling-size invariance (RSI), presents the list encoding, and then proves the RSI properties of list encoding. Section 5 presents concluding remarks and directions for future research.

2 Terminology

The terminology used in this paper is originally from Rau [3] [8], and is introduced here for the discussion that follows. Each wide instruction-word, or *MultiOp*, in a VLIW schedule, consists of several operations, or *Ops*. All Ops in a MultiOp are issued in the same cycle. VLIW programs are *latency-cognizant*, meaning that they are scheduled with the knowledge of functional unit latencies. An architecture which runs latency-cognizant programs is termed a *Non-Unit Assumed Latency* (NUAL) architecture. A *Unit Assumed Latency* (UAL) architecture assumes unit latencies for all functional units. Most superscalar architectures are UAL, whereas most VLIWs are NUAL. The machine models discussed in this paper are NUAL.

There are two scheduling models for latency-cognizant programs: the *Equals* model and the *Less-Than-or-Equals (LTE)* model [9]. Under the equals model, schedules are built such that each operation takes exactly as much as its specified execution latency. In contrast, under the LTE model an operation may take *less than or equal* to its specified latency. In general, the equals model produces slightly shorter schedules than the LTE model; this is mainly due to register re-use possible in the equals model. However, the LTE model simplifies the implementation of precise interrupts and provides binary compatibility when latencies are reduced. Both the scheduler (in the back-end of the compiler) and the dynamic rescheduler (in the page-fault handler) presented in this paper follow the LTE scheduling model.

For the purposes of this paper, it is assumed that all program codes can be classified into two broad categories: acyclic code and cyclic code. Cyclic code consists of short inner loops in the program which typically are amenable to software pipelining [10]. On the other hand, acyclic code contains a relatively large number of conditional branches, and typically has large loop bodies. This makes the acyclic code un-amenable to software pipelining. Instead, the body of the loop is treated as a piece of acyclic code, surrounded by the loop control Ops. Examples of cyclic code are the inner loops like counted DO-loops found

in scientific code. Examples of acyclic code are non-numeric programs, and interactive programs. This distinction between the types of code is made because the scheduling and rescheduling algorithms for cyclic and acyclic code differ considerably, because of which the dynamic rescheduling technique treats each separately.

It is also assumed that the program code is structured in the form of *superblocks* [11] or the *hyperblocks* [12]. Hyperblocks are constructed by if-conversion of code using predication [13], [14]. Support for predicated execution of Ops is also assumed. Both superblocks and hyperblocks have a single entry point into the block (at the beginning of the block) and may have multiple side-exits. This property is useful in bypassing the problems introduced by speculative code motion in DR, discussion of which can be found elsewhere (see [15]).

3 Overview

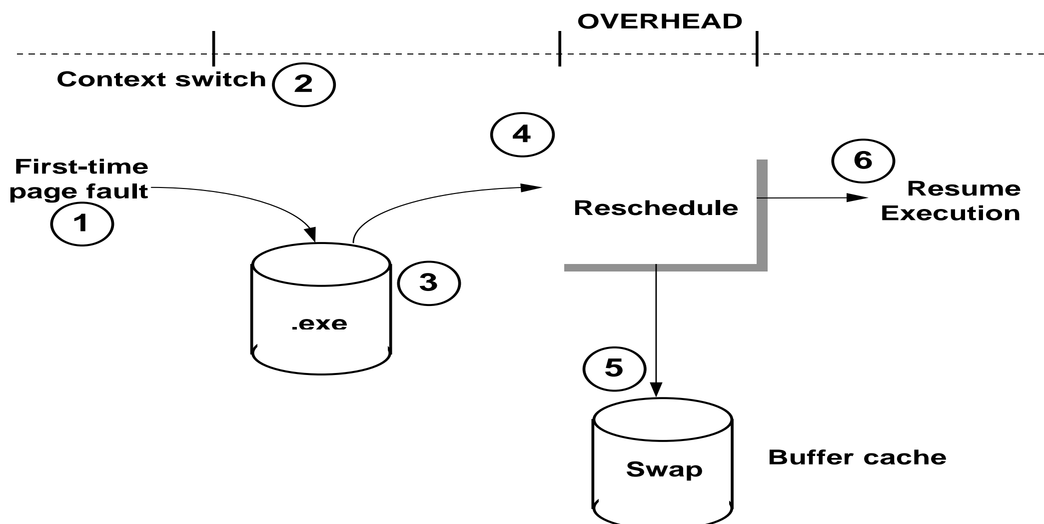

Figure 4: Dynamic Rescheduling: Sequence of events.

The technique of dynamic rescheduling performs translation of code pages at first-time page faults and stores the translated pages for subsequent use. Figure 4 shows the sequence of events that take place in Dynamic Rescheduling. Event 1 indicates a first-time page-fault. On a page-fault, the OS switches context and fetches the requested page from the next level of the memory hierarchy; this is shown shown as Events 2 & 3 respectively. Events 1, 2, 3 are standard in the case of every page fault encountered by the OS. What is different in the case of DR is the invocation of a module called the rescheduler at each first-time page fault. The rescheduler operates on the newly fetched page to reschedule it to execute correctly on the host machine. This is shown as event 4. Event 5 shows that the rescheduled page is written to an area of the file system for future use, and in event 6, the execution resumes.

To facilitate the detection of a VLIW generation mismatch at a first-time page fault, each program binary holds a *generation-id* in its header. The machine model for which the binary was originally scheduled and the boundaries to identify the pieces of cyclic code in

the program are also stored in the program binary. This information is made available to the rescheduler it while performs rescheduling. A page of the rescheduled code remains in the main memory until it is displaced (as any other page in the memory), at which time it is written to a special area of the file system called *text swap*. All subsequent accesses to the page during the lifetime of the program are fulfilled from the text swap. Text swap may be allocated on a per-executable basis at compile time, or be allocated by the OS as a system-wide global area shared by all the active processes. The overhead of rescheduling can be quantitatively expressed in terms of the following factors: (1) the time spent at the first-time page-faults to reschedule the page, (2) the time spent in writing the rescheduled pages to the text swap area, and, (3) the amount of disk space used to store the translated pages. Further discussion of the overhead introduced by DR and an investigation of trade-offs involved in the design of the text-swap used to reduce the overhead are beyond the scope of this paper (see [16] for more details).

3.1 Insertion and deletions of NOPs

When the compiler schedules code for a VLIW, independent Ops which can start execution in the same machine cycle are grouped together to form a single MultiOp; each Op in a MultiOp is bound to execute on a specific functional unit. Often, however, the compiler cannot find enough Ops to keep all the FUs busy in a given cycle. These empty slots in a MultiOp are filled with NOPs. In some machine cycles the compiler cannot schedule even a single Op for execution; NOPs are scheduled for all FUs in such a cycle and the instruction is called as an *empty MultiOp*.

Logically, the rescheduler in DR can be thought of as performing the following steps to generate the new code[1], no matter what the type of code. First, it breaks down each MultiOp into individual Ops, to create an ordered set of Ops. Second, it discards the NOPs from this set. The ordered set of Ops thus obtained is a UAL schedule. In the third step, depending upon the resource constraints and the data dependence constraints, it re-arranges the Ops in the UAL schedule to create the new, NUAL schedule. In the fourth and last step, new NOPs and empty MultiOps are inserted as required to preserve the semantics of the computation. Note that the number of NOPs and the empty MultiOps that are newly inserted may not be the same as that in the old code, which may lead to the problem that the size of the code may change due to rescheduling. It is important to note at this time that the change in the code-size, if any, is only due to the NOPs and the empty MultiOps. An example of changes in code-size is illustrated in Figure 5. In the left portion of the Figure, the old code is shown. Assume that the execution latency of Ops A, D, E, F, G, H is 1-cycle each; that of Op B is 3-cycles and of the LOAD Op C is 2-cycles. Further, Ops E, F are dependent on the result of Op C, hence should not begin execution before Op C finishes execution. In a newer generation of the architecture shown on right, one IALU is removed from the machine, while increasing the latency of the LOADs by 1 (to 3-cycles). When the old code is executed on the newer generation, DR invokes the rescheduler, which generates the new code as shown. To account for the new, longer latency of the LOAD unit, it inserts

[1]The terms "new code" and "old code" do not necessarily mean that the code input to the rescheduler belongs to the *older* machine generation, or similar for the counterpart. "Old code" as used here means any code input to the rescheduler, and new code means the code output by the rescheduler.

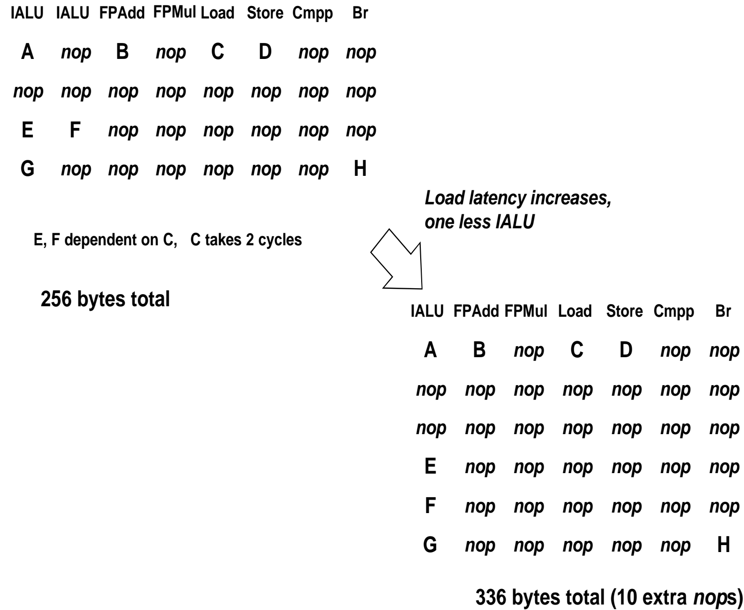

Figure 5: Example illustrating the insertion/deletion of NOPs and empty MultiOps due to Dynamic Rescheduling.

an empty MultiOp in the third cycle. Also, the old MultiOp consisting of operations E and F is broken into two consecutive MultiOps due to the reduction in the number of IALUs. Observe that the new code is bigger than the old code. Assuming all the Ops are 64-bits each, the net increase in the size of the code is 80 bytes, corresponding to the 10-extra NOPs inserted during rescheduling.

The page size with which a computer system operates is usually dictated by the hardware or the OS or both. It is non-trivial for the OS to handle any changes in the page sizes at run-time. Previous work done in this area by Talluri and Hill attempts to support multiple page-sizes, where each page-size is an integral multiple of a base page-size [17] [18] [19]. Enhanced VM hardware (the *Translation-Lookaside Buffer (TLB)*, and an enhanced VM management policy must be available in the to support the proposed technique. It is possible that with the help of this extra hardware, multiple code page sizes can be used to handle variations in page-size due to DR, but this would lead to a multitude of problems. The first problem is that of inefficient memory usage: if a new page is created to accommodate the “spill-over” generated by the rescheduler, the remainder of the new page remains unused. On the other hand, if the code in a page shrinks due to DR, that leads to a hole in the memory. The second problem arises due to control restructuring: when a new page is inserted, it must be placed at the end of the code address space. The last MultiOp in the original page must then be modified to jump to the new page, and the last MultiOp on the new page must be modified to jump to the page which lies after the original page. Now, if a code positioning optimization was performed on the old code in order to optimize for I-Cache accesses, this process could violate the ordering, potentially leading to performance degradation. Perhaps the most serious problem is that the code movement within the old page or into the new

page could alter branch target addresses (merge points) in the old code, leading to incorrect code. It may not even be possible to repair this code, because the code which jumps to the altered branch targets may not be visible to the rescheduler at rescheduling time.

One solution to avoid the problem of code-size change is to use a specialized ISA encoding which "hides" the NOPs and the empty MultiOps in the code. Since all code-size changes in DR are due to the addition/deletion of NOPs, such an encoding circumvents the problem. An encoding called the *list encoding* which has this ability is discussed in detail in Section 4, along with the rescheduling algorithms for cyclic and acyclic code.

4 Rescheduling Size-Invariance

List Encoding is an ISA encoding which does not require explicit representation of NOPs and empty MultiOps in the object code, and hence it is a *zero-NOP* encoding. This property of List encoding is used to support DR. This section presents a formal definition of list encoding, followed by an introduction to the concept of *Rescheduling Size-Invariance (RSI)*. It will also be shown that any list-encoded schedule of code is *rescheduling size-invariant*.

4.1 List encoding and RSI

Definition 1 (Op) *A VLIW Operation (*Op*) is defined by a 6-tuple:* $\{H,\ p_n,\ s_{pred},$ FUtype, opcode, operands$\}$*, where,* $H \in \{0,1\}$ *is a 1-bit field called* header-bit*,* p_n *is an n-bit field called* pause*,* s.t. $p_n \in \{0,\ldots,2^n-1\}$*,* s_{pred} *is a stage predicate (discussed further in Section 4.3),* FUtype *uniquely identifies the FU instance where the* Op *must execute,* opcode *uniquely identifies the task of the* Op*, and* operands *is the set of valid operands defined for* Op*. All* Op*s have a constant width.* □

Definition 2 (Header Op) *An* Op*, O, is a* Header Op *iff the value of the* header-bit *field in O is 1.* □

Definition 3 (VLIW MultiOp) *A VLIW MultiOp, M, is defined as an unordered sequence of* Ops $\{O_1, O_2, \ldots, O_m\}$, s.t. $0 < m \le w$*, where w is the number of hardware resources to which the Ops are issued concurrently, and* O_1 *is a Header Op.* □

Definition 4 (VLIW Schedule) *A VLIW schedule, S, is defined as an ordered sequence of MultiOps* $\{M_1, M_2, \ldots\}$. □

A discussion of the list encoding is now in order. All Ops in this scheme of encoding are fixed-width. In a given VLIW schedule, a new MultiOp begins at a Header Op and ends exactly at the Op before the next Header Op; the MultiOp fetch hardware uses this rule to identify and fetch the next MultiOp. The value of the p_n field in an Op is referred to as the *pause*, because it is used by the fetch hardware to stop MultiOp fetch for the number of machine cycles indicated by p_n. This is a mechanism devised to eliminate the explicit encoding of empty MultiOps in the schedule. The *FUtype* field indicates the functional unit where the Op will execute. The *FUtype* field allows the elimination of NOPs inserted by

the compiler in an arbitrary MultiOp. Prior to the execution of a MultiOp, its member Ops are routed to their appropriate functional units based on the value of their *FUtype* field.

This scheme of encoding the components of a VLIW schedule is termed as *List encoding.* Since the size of every Op is the same, the size of a given list encoded schedule, S, can be expressed in terms of the number of Ops in it:

$$sizeof(S) = \sum_i \sum_j O_j$$

where i is the number of MultiOps in S, O_j is an Op, and j is the number of Ops in a given MultiOp.

Definition 5 (VLIW Generation) *A VLIW* generation *G is defined by the set $\{R, L\}$, where R is a set of hardware resources in G, and $L \in \{1, 2, \ldots\}$ is the set of execution latencies of all the Ops in the operation set of G. R itself is a set consisting of pairs $\{r, n_r\}$, where r is a* resource type *and n_r is the number of instances of r.*

This definition of a VLIW generation does not model complex resource usage patterns for each Op, as used in [20], [21] and [22]. Instead, each member of the set of machine resources R, presents a higher-level abstraction of the "functional units" found in modern processors. Under this abstraction, the low-level machine resources such as the register-file ports and operand/result busses required for the execution of an Op on each functional unit are bundled with the resource itself. All the resources indicated in this manner are assumed to be busy through the period of time equal to the latency of the executing Op, indicated by the appropriate member of set L.

Definition 6 (Rescheduling Size-Invariance (RSI)) *A VLIW schedule S is said to satisfy the RSI property iff* $\text{sizeof}(S_{G_n}) = \text{sizeof}(S_{G_m})$*, where S_{G_n}, S_{G_m} are the versions of the original schedule S prepared for execution on arbitrary machine generations G_n and G_m respectively. Further, schedule S is said to be* rescheduling size-invariant *iff it satisfies the RSI property.* □

The proof that list encoding is RSI will be presented in two parts. First, it will be shown that acyclic code in the program is RSI when list encoded, followed by the proof that the cyclic code is RSI when list encoded. Since all code is assumed to be either acyclic or cyclic, the result that list encoding makes it RSI will follow. In the remainder of this section, algorithms to reschedule each of these types of codes are presented, followed by the proofs themselves.

4.2 Rescheduling Size-Invariant Acyclic Code

The algorithm to reschedule acyclic code from VLIW generation G_{old} to generation G_{new} is shown in Algorithm *Reschedule_Acyclic_Code.* It is assumed that both the old and new schedules are LTE schedules (see Section 2), and that both have the same register file architecture and compiler register usage convention.

Algorithm *Reschedule_Acyclic_Code*

input

S_{old}, the old schedule, (assumed no more than n_{old} cycles long);
$G_{old} = \{R_{old}, L_{old}\}$, the machine model description for the *old* VLIW;
$G_{new} = \{R_{new}, L_{new}\}$, the machine model description for the *new* VLIW;

output

S_{new}, the new schedule;

var

n_{old}, the length of S_{old};
n_{new}, the length of S_{new};
$Scoreboard$[number of registers], to flag the registers "in–use" in S_{old}.
$RU[n_{new}][\sum_r n_r]$, the resource usage matrix, where:
  r represents all the resource types in G_{new}, and,
  n_r is the number of instances of each resource type r in G_{new};
UseInfo$[n_{new}]$[number of registers], to mark the register usage in S_{new};
T_δ, the cycle in which an Op can be scheduled while
  satisfying the data dependence constraints;
$T_{rc+\delta}$, the cycle in which an Op can be scheduled while satisfying
  the data dependence and resource constraints;

functions

RU_loopkup $(O\,(T_\delta))$ returns the earliest cycle, later than cycle T_δ, in which Op O
  can be scheduled after satisfying the data dependence and resource constraints;
RU_update (σ, O) marks the resources used by Op O in cycle σ of S_{new};
dest_register (O) returns the destination register of Op O;
source_register (O) returns a list of all source registers of Op O;
latest_use_time (ϕ) returns the latest cycle in S_{new} that register ϕ was used in;
most_recent_writer (ρ) returns the id of the Op which modified register ρ latest in S_{old};

begin

for each MultiOp $M_{old}[c] \in S_{old}$, $0 \le c \le n_{old}$ **do**
begin
  — *resource constraint check:*
  for each Op $O_w \in S_{old}$ that completes in cycle c **do**
    begin
      $O_w\,(T_{rc+\delta}) \leftarrow RU_loopkup\,(O_w\,(T_\delta))$;
      $M_{new}[new_cycle] \leftarrow M_{new}[new_cycle] \mid O_w$;
      $RU_update\,(T_{rc+\delta}, O_w)$;
    end
  — *update the scoreboard:*
  for each Op $O_i \in S_{old}$ which is unfinished in cycle c **do**
    begin
      $Scoreboard[dest_register\,(O_r)] \leftarrow reserved$;
    end
  — *do data dependence checks:*

```
        for each Op O_r ∈ M_old[c] do
          begin
            O_r (T_δ) ← 0;
            —  anti-dependence:
            for each φ ∈ dest_register (O_r) do
                O (T_δ) ← MAX(O (T_δ), latest_use_time (φ));
            —  pure dependence:
            for each φ ∈ source_register (O_r) do
              if (Scoreboard[φ] = reserved)
                O_r (T_δ) ← MAX (O_r(T_δ), completion_time (reserving_Op));
            —  output dependence:
            for each φ ∈ dest_register (O_r) do
              if (Scoreboard[φ] = reserved)
                O_r (T_δ) ← MAX (O_r(T_δ), completion_time (reserving_Op));
            end
    end
end
```

The RSI property for list encoded acyclic code schedule will now be proved.

Theorem 1 *An arbitrary list encoded schedule of acyclic code is RSI.*

Proof: The proof will be presented using induction over the number of Ops in an arbitrary list encoded schedule. Let L_i be an arbitrary, ordered sequence of i Ops ($i \geq 1$) that occur in a piece of acyclic code. Let F_i denote a directed dependence graph for the Ops in L_i, *i.e.* each Op in L_i is a node in F_i, and the data- and control-dependences between the Ops are indicated by directed arcs in F_i. Let S_{G_n} be the list encoded schedule for L_i generated using the dependence graph F and designed to execute on a certain VLIW generation G_n. Also, let G_m denote another VLIW generation which is the target of rescheduling under DR.

Induction Basis. L_1 is an Op sequence of length 1. In this case, $\textit{sizeof}\,(S_{G_n}) = 1$, and the dependence graph has a single node. It is trivial in this case that S_{G_n} is RSI, because after rescheduling to generation G_m, the number of Ops in the schedule will remain 1, or,

$$\textit{sizeof}\,(S_{G_n}) = \textit{sizeof}\,(S_{G_m}) = 1 \tag{1}$$

Induction Step. L_p is an Op sequence of length p, where $p > 1$. Assume that S_{G_n} is RSI. In other words,

$$\textit{sizeof}\,(S_{G_n}) = \textit{sizeof}\,(S_{G_m}) = p \tag{2}$$

Now consider the Op sequence L_{p+1}, which is of length $p+1$, such that to L_{p+1} was obtained from L_p by adding one Op from the original program fragment. Let this additional Op be denoted by z. Op z can be thought of as borrowed from the original program, such that the correctness of the computation is not compromised. L_p is an ordered sequence of Ops, and Op z must then be either a prefix of L_p, or a suffix to it. Also, let T_{G_n} denote the list encoded schedule for sequence L_{p+1}, which means $\textit{sizeof}\ (T_{G_n}) = p + 1$. In order to prove the current theorem, it must now be proved that T_{G_n} is RSI if S_{G_n} is RSI.

The addition of Op z to L_p may change the structure of the dependence graph F_p in two ways: (1) if the Op z adds one or more data dependence arcs to F_p, or (2) the Op z does not add any data dependence arcs to F_p.

- **Op z adds dependence(s)**:
  This case corresponds to the fact that Op z is control- and/or data-dependent on one or more of the Ops in L_p, or *vice versa*. Following are the two sub-cases in which a schedule will be constructed which includes the Op z: (1) construction of T_{G_n} using the dependence graph F, and, (2) rescheduling of T_{G_n} to T_{G_m}. In both these cases, all the dependences introduced by Op z must be honored. Further, any resource constraints must be satisfied as well. This is done using the well-known list scheduling algorithm (in the first sub-case), and the *Reschedule_Acyclic_Code* algorithm (in the second sub-case). Appropriate NOPs and empty MultiOps will be inserted in the schedule by both these algorithms. However, when the schedules T_{G_n} and T_{G_m} are list encoded, the empty MultiOps will be made implicit using the *pause* field in the Header Op of the previous MultiOp, and the NOPs in a MultiOp will be made implicit via the *FUtype* field in the Ops. Thus, the only source of size increase in schedules T_{G_n} and T_{G_m} is due to the newly added Op z.

- **Op z does not add any dependences**:
  In this case, only the resource constraints, if any, would warrant the insertion of empty MultiOps. By an argument similar to that in the previous case, it is trivial to see that the only source of size increase in schedules T_{G_n} and T_{G_m} is the newly added Op z.

Thus, in both the cases, $sizeof(T_{G_n}) = sizeof(S_{G_n}) + 1$, from which and from Equation 2, it follows that:

$$sizeof(T_{G_n}) = p + 1 \tag{3}$$

Similarly, for both the cases, $sizeof(T_{G_m}) = sizeof(S_{G_m}) + 1$, which leads to:

$$sizeof(T_{G_m}) = p + 1 \tag{4}$$

From Equations 3 and 4, and by induction, it is proved that an arbitrary list encoded schedule of acyclic code is RSI. ■

An example of the transition of the code previously shown in Figure 5, by application of algorithm *Reschedule_Acyclic_Code* is shown in Figure 6 (assuming that the original schedule belonged to the *acyclic* category). It can be observed that the size of the original code (on the left) is the same as that of the rescheduled code (on the right). The NOPs and the empty MultiOps have been eliminated in the *list encoded* schedules; the rescheduling algorithm merely re-arranged the Ops, and adjusted the values of the H and the p_n (pause) fields within the Ops to ensure the correctness of execution on G_{new}.

4.3 Rescheduling Size-Invariant Cyclic Code

Most programs spend a great deal of time executing the inner loops, and hence the study of scheduling strategies for inner loops has attracted great attention in literature [23], [24], [25], [8], [26], [27], [28], [29]. Inner loops typically have small bodies (relatively fewer Ops) which

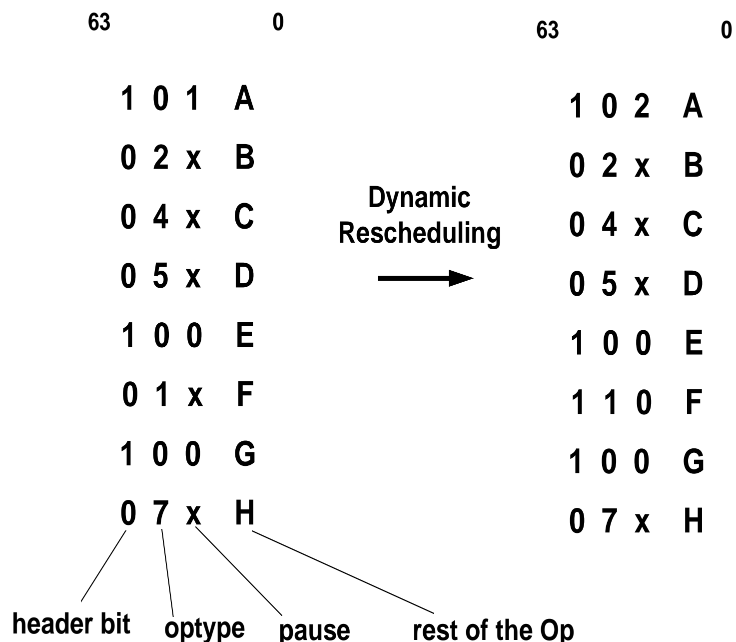

Figure 6: Example: List encoded schedule for acyclic code is RSI.

makes it hard to find ILP within these loop-bodies. Software pipelining is a well-understood scheduling strategy used to expose the ILP across multiple iterations of the loop [30], [25]. There are two ways to perform software pipelining. The first one uses *loop unrolling*, in which the loop body is unrolled a fixed number of times before scheduling. Loop bodies scheduled via unrolling can be subjected to rescheduling via the *Reschedule_Acyclic_Code* algorithm described in Section 4.2. The code expansion introduced due to unrolling, however, is often unacceptable, and hence the second technique, *Modulo Scheduling* [30], is employed. Modulo-scheduled loops have very little code expansion (as the *prologue* and *epilogue* of the loop) which makes it very attractive. In this paper, only modulo-scheduled loops are examined for the RSI property; unrolled-and-scheduled loops are covered by the acyclic RSI techniques presented previously. First, some discussion of the structure of modulo-scheduled loops in presented, followed by an algorithm to reschedule modulo scheduled code. The section ends with a formal treatment to show the list-encoded modulo-scheduled cyclic code is RSI. Concepts from Rau [29] are used as a vehicle for the discussion in this section.

Assumptions about the hardware support for execution of modulo scheduled loops are as follows. In some loops, a datum generated in one iteration of the loop is consumed in one of the successive iterations (an inter-iteration data dependence). Also, if there is any conditional code in the loop body, multiple, data-dependent paths of execution exist. Modulo-scheduling such loops is non-trivial[2]. This paper assumes three forms of hardware support to circumvent these problems. First, register renaming via *rotating registers* [29] in order to handle the inter-iteration data dependencies in loops is assumed. Second, to convert the control dependencies within a loop body to data dependencies, support for *predicated*

[2]See [31] and [32] for some of the work in this area.

execution [14] is assumed. Third, support for *sentinel scheduling* [33] to ensure correct handling of exceptions in speculative execution is assumed. Also, the *pre-conditioning* [29] of counted-DO loops is presumed to have been performed by the modulo scheduler when necessary.

A modulo scheduled loop, Ω_{G_n}, consists of three parts: a *prologue* (π_{G_n}), a *kernel* (κ_{G_n}), and an *epilogue* (ε_{G_n}), where G_n is the machine generation for which the loop was scheduled. The prologue initiates a new iteration every II cycles, where II is known as the *initiation interval*. Each slice of II cycles during the execution of the loop is called a *stage*. In the last stage of the first iteration, execution of the kernel begins. More iterations are in various stages of their execution at this point in time. Once inside the kernel, the loop executes in a steady state (so called because the kernel code branches back to itself). In the kernel, multiple iterations are simultaneously in progress, each in a different stage of execution. A single iteration completes at the end of each stage. The branch Ops used to support the modulo scheduling of loops have special semantics, by which the branch updates the loop counts and enables/disables the execution of further iterations. When the loop condition becomes false, the kernel falls through to the epilogue, which allows for the completion of the stages of the unfinished iterations. Figure 7 shows an example modulo schedule for a loop and identifies the prologue, kernel, and the epilogue. Each row in the schedule describes a cycle of execution. Each box represents a set of Ops that execute in a same resource (e.g. functional unit) in one stage. The height of the box is the II of the loop. All stages belonging to a given iteration are marked with a unique alphabet $\in \{A, B, C, D, E, F\}$.

Figure 7 also shows the loop in a different form: the *kernel-only (KO)* loop [26] [29]. In a kernel-only loop, the prologue and the epilogue of the loop "collapse" into the kernel, without changing the semantics of execution of the loop. This is achieved by predicating the execution of each distinct stage in a modulo scheduled loop on a distinct predicate called a *stage predicate*. A new stage predicate is asserted by the loop-back branch. Execution of the stage predicated on the newly asserted predicate is enabled in the future executions of the kernel. When the loop execution begins, stages are incrementally enabled, accounting for the loop prologue. When all the stages are enabled, the loop kernel is in execution and the loop is in the steady state. When the loop condition becomes false, the predicates for the stages are reset, thus disabling the stages one by one. This accounts for the iteration of the epilogue of the loop. A modulo scheduled loop can be represented in the KO form, if adequate hardware (predicated execution) and software (a modulo scheduler to predicate the stages of the loop) support is assumed. Further discussion of KO loop schedules can be found in [29]. All modulo-scheduled loops can be represented in the KO form. The KO form thus has the potential to encode modulo schedules for all classes of loops, a property which is useful in the study of dynamic rescheduling of loops, as will be shown shortly.

The size of a modulo scheduled loop is larger than the original size of the loop, if the modulo schedule has an explicit prologue, a kernel, and an epilogue. In contrast, a KO loop schedule has exactly one copy of each stage in the original loop body, and hence has the same size as the original loop body, provided the original loop was completely if-converted[3]. This property of the KO loops is useful in performing dynamic rescheduling of modulo scheduled

[3]For any *pre-conditioned* counted DO-loops, the size is the same as the size of loop body after pre-conditioning.

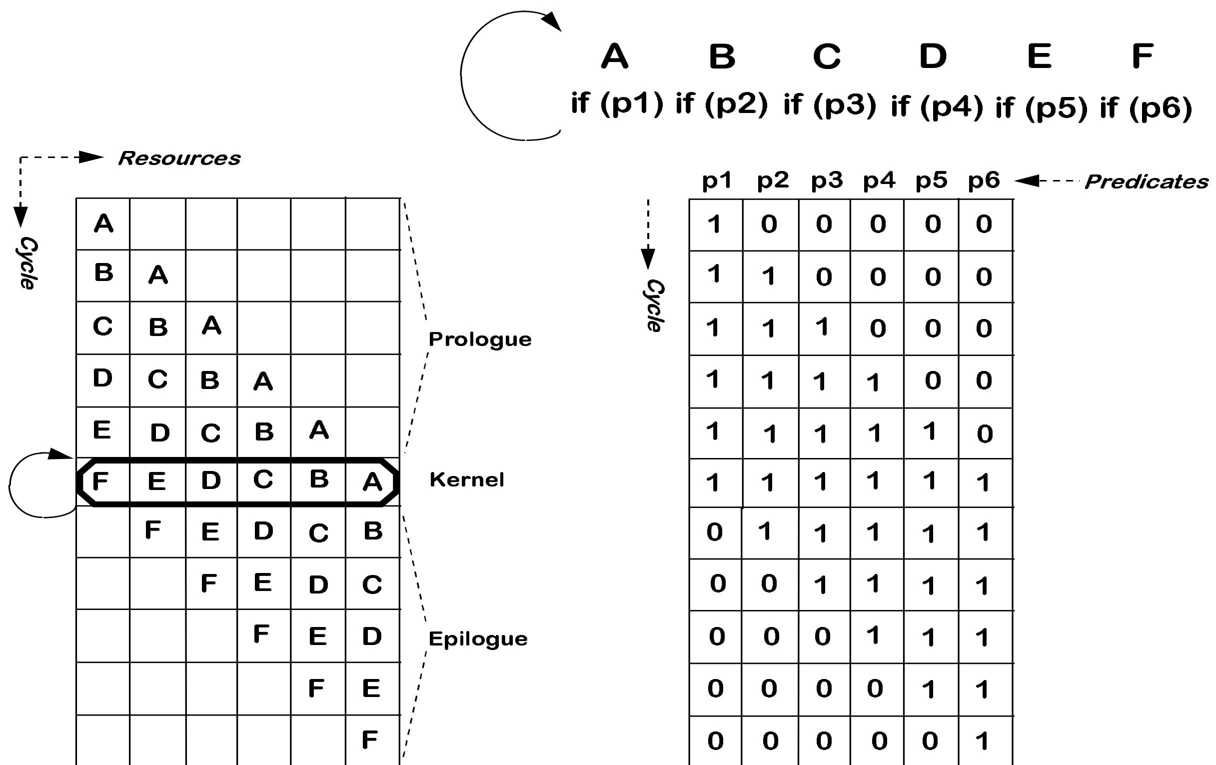

Figure 7: A Modulo scheduled loop: on the left, a modulo scheduled loop (with the *prologue*, *kernel*, and the *epilogue* marked) is shown. The same schedule is shown on the right, but in the "collapsed" *kernel-only (KO)* form. *Stage predicates* are used to turn the execution of the Ops ON or OFF in a given stage. The table shows values that the stage predicates $\{p1, p2, p3, p4, p5, p6\}$ would take for this loop.

loops. Algorithm *Reschedule_KO_Loop* details the steps. The input to the algorithm is the modulo scheduled KO loop, and the machine models for the old and the new generations G_{old} and G_{new}. Briefly, the algorithm works as follows: identification of the predicates that enable individual stages is performed first. An order is imposed on them, which then allows for the derivation of the order of execution of stages in a single iteration. The ordering on the predicates may be implicit in the predicate-id used for a given stage (increasing order of predicate-ids). Alternatively, the order information could be stored in the object file and made available at the time DR is invoked, without substantial overhead. Once the order of execution of the stages of the loop is obtained, the reconstruction of the loop in its original, unscheduled form is performed. At this time, the modulo scheduler is invoked on it to arrive at the new KO schedule for the new generation.

Algorithm *Reschedule_KO_Loop*

input

Ω_{old} =the KO (kernel-only) modulo schedule such that:
number of stages = n_{old}; initiation interval = II_{old};
$G_{old} = \{R_{old}, L_{old}\}$, the machine model description for the *old* VLIW;
$G_{new} = \{R_{new}, L_{new}\}$, the machine model description for the *new* VLIW;

output

Ω_{new}: KO (kernel-only) modulo schedule for G_{new};

var

$B[n_{old}]$, the table of n_{old} buckets each holding the Ops from a unique stage, such that the relative ordering of Ops in the bucket is retained;

functions

FindStagePred (O) returns the stage predicate on which Op O is enabled or disabled;
BucketOP $(O, B[p])$ puts the Op O into the bucket $B[p]$;
OrderBuckets $(B, func)$ sorts the table of buckets B according to the ordering function *func*;
StagePredOrdering () describes the statically imposed order on the stage predicates;

begin

— *unscramble the old modulo schedule:*
for all MultiOps $M \in \Omega_{old}$ **do**
  for each Op $O \in M$
    begin
      $p =$ *FindStagePred* (O);
      BucketOP $(O, B[p])$;
    end
— *order the buckets:*
OrderBuckets (*B*, *StagePredOrdering* ());
— *perform modulo scheduling:*
Perform modulo scheduling on the sorted table of buckets B, using the algorithm described by Rau in [30] to generate KO schedule Ω_{new};

end

The RSI nature of List encoded modulo scheduled KO loop will now be proved.

Theorem 2 *An arbitrary List encoded Kernel-Only modulo schedule of a loop is RSI.*

Proof: Let L_i be an arbitrary, ordered sequence of i Ops ($i \geq 1$) that represents the loop body. Let F_i denote a directed dependence graph for the Ops in L_i, *i.e.* each Op in L_i is a node in F_i, and the data- and control-dependences between the Ops are indicated by directed arcs in F_i. Note that the inter-iteration data dependences are also indicated in F_i. Let Ω_{G_n} denote a list encoded KO modulo schedule for generation G_n. Also, let G_m denote the VLIW generation for which rescheduling is performed.

Induction Basis. L_1 is a loop body of length 1. In this case, $sizeof(\Omega_{G_n}) = 1$, and the dependence graph has a single node. It is trivial in this case that Ω_{G_n} is RSI, because after rescheduling to generation G_m, the number of Ops in the schedule will remain 1, or,

$$sizeof(\Omega_{G_n}) = sizeof(\Omega_{G_m}) = 1 \tag{5}$$

(Note that a loop where $sizeof(\Omega_{G_n}) = 1$ is the degenerate case).

Induction Step. L_p is a loop body of length p, where $p > 1$. Assume that Ω_{G_n} is RSI. In other words,

$$sizeof(\Omega_{G_n}) = sizeof(\Omega_{G_m}) = p \tag{6}$$

Now consider another loop body L_{p+1}, which is of length $p+1$. Let $(p+1)^{st}$ Op be denoted by z. Also, let Θ_{G_n} denote the list encoded KO modulo schedule for L_{p+1}, which means *sizeof* $(\Theta_{G_n}) = p + 1$. In order to prove the theorem at hand, it must now be proved that Θ_{G_n} is RSI if Ω_{G_n} is RSI.

It is possible that due to Op z in L_{p+1}, the nature of the graph F_p could be different from that of the graph F_{p+1} in two ways: (1) Op z is data dependent on one or more Ops in L_{p+1} or *vice versa*, or (2) the Op z is independent of all the Ops in L_{p+1}. In both of these cases, the data dependences and the resource constraints are honored by the modulo scheduling algorithm via appropriate use of NOPs and/or empty MultiOps within the schedule. When this schedule is list encoded, the NOPs and the empty MultiOps are made implicit via the use of *pause* and the *FUtype* fields within the Ops. Hence,

$$sizeof(\Theta_{G_n}) - sizeof(\Omega_{G_n}) = sizeof(z) = 1 \tag{7}$$

In other words,

$$sizeof(\Theta_{G_n}) = sizeof(\Omega_{G_n}) + 1 \tag{8}$$

From this result and from Equation 6, it follows that:

$$sizeof(\Theta_{G_n}) = p + 1 \tag{9}$$

Similarly, for both the cases, $sizeof(\Theta_{G_m}) = sizeof(\Omega_{G_m}) + 1$, which leads to:

$$sizeof(\Theta_{G_m}) = p + 1 \tag{10}$$

From Equations 9 and 10, and by induction, it is proved that an arbitrary list encoded KO modulo schedule is RSI.

■

Corollary 1 *A List encoded schedule is RSI.*

Proof: All program codes can be divided into the two categories: the acyclic code and cyclic code as defined in Section 2. Hence, It follows from Theorem 1 and Theorem 2 that a list encoded schedule is RSI. ■

5 Conclusions

This paper has presented the highlights of a solution for the cross-generation compatibility problem in VLIW architectures. The solution, called Dynamic Rescheduling, performs rescheduling of program code pages at first-time page faults. Assistance from the compiler, the ISA, and the OS is required for dynamic rescheduling. During the process of rescheduling, NOPs must be added to/deleted from the page to ensure the correctness of the schedule. Such additions/deletions could lead to changes in the page size. The code-size changes are hard to handle at run-time, would and require extra support in hardware (TLB extensions) and software (VM management extension).

An ISA encoding called List Encoding, which encodes the NOPs in the program implicitly, was presented. The list encoded ISA has fixed-width Ops. The Header Op (first Op) in a MultiOp indicates the number of empty MultiOps (if any) following it. This information eliminates the need to explicitly encode the empty MultiOps in the schedule. The OpType field encoded in each Op eliminates the need of explicitly encoding the NOPs within the MultiOp, because the decode hardware can use this information to expand and route the Op to appropriate execution resource. A property of the list encoding called Rescheduling-Size Invariance (RSI) was proved for the acyclic and cyclic (for kernel-only modulo-scheduled) codes. A schedule of code is RSI iff the code size remains constant across the dynamic rescheduling transformation.

A study of the instruction fetch hardware and I-Cache organizations required to support the list encoding has previously been studied [34]. The work presented in this paper can be extended with a study of other encoding techniques which may not be rescheduling-size invariant (non-RSI encodings). Also, a study of rescheduling algorithms which operate on non-RSI encodings can be conducted. These topics are currently being investigated by the authors.

References

[1] T. M. Conte and S. W. Sathaye, "Dynamic rescheduling: A technique for object code compatibility in VLIW architectures," in *Proc. 28th Ann. Int'l Symp. on Microarchitecture*, (Ann Arbor, MI), Nov. 1995.

[2] J. S. O'Donnell, "Superscalar vs. VLIW," *Computer Architecture News (ACM SIGARCH)*, pp. 26–28, Mar. 1995.

[3] B. R. Rau, "Dynamically scheduled VLIW processors," in *Proc. 26th Ann. Int'l Symp. on Microarchitecture*, (Austin, TX), pp. 80–90, Dec. 1993.

[4] S. Melvin, M. Shebanow, and Y. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proc. 21th Ann. Int'l Symp. on Microarchitecture*, (San Diego, CA), pp. 60–66, Dec. 1988.

[5] G. Silberman and K. Ebcioğlu, "An architectural framework for supporting heterogeneous instruction–set architectures," *Computer*, vol. 26, pp. 39–56, June 1993.

[6] M. Franklin and M. Smotherman, "A fill-unit approach to multiple instruction issue," in *Proc. 27th Ann. Int'l Symp. on Microarchitecture*, (San Jose, CA), pp. 162–171, Dec. 1994.

[7] G. Silberman and K. Ebcioğlu, "An architectural framework for migration from CISC to higher performance platforms," in *Proc. 1992 Int'l Conference on Supercomputing*, pp. 198–215, 1992.

[8] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *Computer*, vol. 22, pp. 12–35, Jan. 1989.

[9] V. Kathail, M. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, Feb. 1994.

[10] A. E. Charlesworth, "An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family," *Computer*, vol. 14, pp. 18–27, Sept. 1981.

[11] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective structure for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1993.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the Hyperblock," in *Proc. 25th Ann. Int'l. Symp. on Microarchitecture*, (Portland, OR), pp. 45–54, Dec. 1992.

[13] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proc. 10th Ann. ACM Symp. on Principles of Programming Languages*, Jan. 1983.

[14] J. C. H. Park and M. Schlansker, "On predicated execution," Tech. Rep. HPL-91-58, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, 1991.

[15] T. M. Conte and S. W. Sathaye, "Optimization of VLIW compatibility systems employing dynamic rescheduling." Accepted for publication in International Journal of Parallel Processing.

[16] T. M. Conte, S. W. Sathaye, and S. Banerjia, "A Persistent Rescheduled–Page Cache for low-overhead object-code compatibility in VLIW architectures," in *Proc. 29th Ann. Int'l Symp. on Microarchitecture* [35].

[17] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in supporting two page sizes," in *Proc. 19th Ann. Int'l Symp. Computer Architecture*, (Gold Coast, Australia), May 1992.

[18] Y. A. Khalidi, M. Talluri, M. N. Nelson, and D. Williams, "Virtual memory support for multiple page sizes," Tech. Rep. TR-93-17, SUN Microsystems Laboratories, Sunnyvale, CA, 1993.

[19] M. Talluri and M. D. Hill, "Surpassing the TLB performance of superpages with less operating system support," in *Proc. 6th Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems*, (Orlando, FL), pp. 171–182, June 1994.

[20] A. E. Eichenberger and E. Davidson, "A reduced multipipeline machine description that preserves scheduling constraints," in *Proc. of the Conference on Programming Language Design and Implementation*, (Philadelphia, PA), May 1996.

[21] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau, "Optimization of machine descriptions for efficient use," in *Proc. 29th Ann. Int'l Symp. on Microarchitecture* [35].

[22] V. Bala and N. Rubin, "Efficient instruction scheduling using finite state automata," in *Proc. 28th Ann. Int'l Symp. on Microarchitecture*, (Ann Arbor, MI), Nov. 1995.

[23] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proc. 14th Annual Workshop on Microprogramming*, pp. 183–198, Nov. 1981.

[24] B. Rau, C. Glaeser, and R. Picard, "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," in *Proc. 9th Ann. Int'l Symp. Computer Architecture*, (Austin, TX), pp. 131–139, Apr. 1982.

[25] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 318–327, June 1988.

[26] J. C. Denhert, P.-T. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proc. Third Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, (Boston, MA), pp. 26–38, Apr. 1989.

[27] A. Nicolau and R. Potasman, "Realistic scheduling: compaction for pipelined architectures," in *Proc. 23rd Ann. Int'l Symp. on Microarchitecture*, (Orlando, FL), pp. 69–79, 1990.

[28] B. Su and J. Wang, "GURPR*: A new global software pipelining algorithm," in *Proc. 24th Ann. Int'l. Symp. on Microarchitecture*, (Albuquerque, NM), pp. 212–216, Nov. 1991.

[29] B. R. Rau, M. Schlansker, and P. P. Tirumalai, "Code generation schemas for modulo scheduled DO-loop and WHILE-loops," Tech. Rep. HPL-92-47, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, 1992.

[30] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Ann. Int'l Symp. on Microarchitecture*, (San Jose, CA), Dec. 1994.

[31] N. Warter, *Modulo Scheduling with Isomorphic Control Transformations.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 1994.

[32] M. G. Stoodley and C. G. Lee, "Software pipelining of loops with conditional branches," in *Proc. 29th Ann. Int'l Symp. on Microarchitecture* [35], pp. 262–273.

[33] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: A model for compiler-controlled speculative execution," *ACM Trans. Comput. Sys.*, vol. 11, pp. 376–408, Nov. 1993.

[34] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction fetch mechanisms for VLIW architectures with compressed encodings," in *Proc. 29th Ann. Int'l Symp. on Microarchitecture* [35].

[35] *Proc. 29th Ann. Int'l Symp. on Microarchitecture*, (Paris, France), Dec. 1996.