Automatic Instruction Scheduler Retargeting by Reverse-Engineering

Matthew J. Bridges Neil Vachharajani Guilherme Ottoni David I. August

Department of Computer Science
Princeton University
{mbridges,nvachhar,ottoni,august}@princeton.edu

Abstract

In order to generate high-quality code for modern processors, a compiler must aggressively schedule instructions, maximizing resource utilization for execution efficiency. For a compiler to produce such code, it must avoid structural hazards by being aware of the processor's available resources and of how these resources are utilized by each instruction. Unfortunately, the most prevalent approach to constructing such a scheduler, manually discovering and specifying this information, is both tedious and error-prone.

This paper presents a new approach which, when given a processor or processor model, automatically determines this information. After establishing that the problem of perfectly determining a processor's structural hazards through probing is not solvable, this paper proposes a heuristic algorithm that discovers most of this information in practice. This can be used either to alleviate the problems associated with manual creation or to verify an existing specification. Scheduling with these automatically derived structural hazards yields almost all of the performance gain achieved using perfect hazard information.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering; D.3.4 [*Programming Languages*]: Processors—Retargetable compilers

General Terms Measurement, Experimentation, Verification, Algorithms

Keywords reverse-engineering, instruction scheduling, compilers, automatic retargeting, structural hazard

1. Introduction and Motivation

Instruction scheduling, an important optimization in modern compilers, attempts to minimize the execution time for a set of instructions by orchestrating the order of their execution. Scheduling is particularly important for wide-issue machines, where instruction level parallelism (ILP) is a key source of performance, as it is responsible for presenting sets of instructions for concurrent execution. A naïve instruction scheduler that does not respect resource limitations may present the hardware with data-independent operations unable to execute concurrently due to *structural hazards*, the over-subscription of limited machine resources. Thus, to be effective, an instruction scheduler must not only be aware of data dependences but also of the processor's resource limitations [19, 25].

PLDI'06 June 10–16, 2006, Ottowa, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006...$5.00.

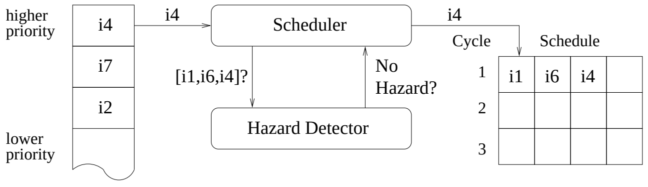

Figure 1. Basic Scheduling Algorithm

Figure 1 shows how a typical scheduler might avoid structural hazards using a *hazard detector* during scheduling. In each step, the scheduler proposes adding an instruction to the existing schedule producing a candidate schedule. The instruction is chosen based on a heuristic function which weighs several concerns, including the belief that the data dependences for the instruction will be satisfied in this cycle in the final schedule. The hazard detector is queried by the instruction scheduler with the candidate schedule and determines whether this schedule now contains a structural hazard. If it does not, the candidate schedule is kept. Otherwise, the scheduler will respect the structural hazard and look for other scheduling alternatives to propose.

Hazard detectors traditionally use resources encoded in *resource maps* [8, 10, 15, 17], which describe the resources consumed by an instruction as it flows through the processor. Using resource maps, a hazard detector can determine when a structural hazard, or *conflict*, will occur. The diversity of instructions in instruction set architectures (ISA), coupled with numerous asymmetric resources in modern microarchitectures, yields large and complex resource maps. Consequently, manually describing the resource maps of an ISA can be a tedious and error prone process [26].

To automate resource map generation, techniques based on a formal processor model have been proposed [16, 34]. These require that the model be developed in architecture description languages (ADL) such as TDL, LISA, etc. [23, 36, 37]. Unfortunately, in practice, ADLs are not useful to compiler writers. The most common problem for compiler writers is that an ADL specification may not be available for the target processor. There are several reasons for this. First, there is no standard ADL. Of those that have been proposed, none can accurately describe the wide class of architectures used today [27]. Second, while ADL descriptions do exist for some machines, the languages themselves often include assumptions about a set of machines. This means that the final circuit-level design constraints often deviate from the ADL specification and the abilities of the ADL itself, making the ADL inaccurate. Finally, even if an accurate ADL were to exist for a given machine, it might be kept as a trade secret. For most processors, these problems mean that compiler writers must look to other approaches for resource map generation.

The solution most commonly pursued by compiler writers is to manually create a *machine description file* [4, 8, 15] to provide hazard detectors with machine-specific resource maps. These are built from publicly available information, usually in the form of instruction and processor reference manuals. The accuracy of the resulting resource maps is dependent upon both the accuracy of the reference manuals and the thoroughness of the compiler writer who interprets these manuals. Unfortunately, the manuals are often complex, hard to read, and contain inaccuracies or contradictions. For example, the Itanium 2 processor manual [22] and microarchitecture manual [21] make contradictory statements regarding the resolution of bank conflicts in the L2 data cache. The former states that a 7 cycle latency for the data to be retrieved, while the latter states an 11 cycle latency.

Even when the manuals are correct, the compiler writer may make a mistake when writing the description. High-level machine description languages have been proposed [17, 23, 37] to alleviate this problem by factoring descriptions into simpler parts. However, creating these descriptions is time-consuming since the writer must still model the inherently complex interactions present in a microarchitecture [9, 16]. For example, the Itanium 2 IMPACT machine description, written in the high-level IMPACT MDes, takes 2700 lines to describe 309 instructions. Even though resource utilization in the Itanium 2 can be broken down into categories, the bundling constraints, unique to this microarchitecture, are 800 lines alone and took approximately a month to add [20]. Even after many years of use, this description still contains several errors as shown in Section 6.4.

Since ADL or other formal descriptions often do not exist, the manuals are often incomplete or inaccurate, and construction by hand is error prone, perhaps structural hazard information could be automatically extracted from the processor itself. The contribution of this paper is to show that such an approach is viable. This paper describes and demonstrates the previously unexplored approach of automatically reverse-engineering structural hazards from a real processor or, in general, any machine model. The technique is compatible with any processor design methodology and tool chain, and it only requires a machine or simulation model that can return the execution time of a program executed on it. While this paper proves that automatically reverse-engineering *all* structural hazards by observing only the machine's behavior is impossible in all cases, it also demonstrates that a heuristic-based approach can determine *almost all* structural hazards in practice. This automatically determined structural hazards can be used to alleviate the problems associated with manual creation or to verify an existing specification. We show that scheduling with these automatically derived structural hazards yields almost all of the performance gain achieved using perfect hazard information. Additionally, our technique suggests hybrid approaches in which manual specification of resource maps is validated by automatic reverse-engineering and vice versa.

The rest of the paper is organized as follows. Section 2 discusses work related to resource maps, hazard detection, and automatic compiler construction. Section 3 describes the problem of automatic determination of structural hazards. Practical observations used to reduce the search space are given in Section 4. These observations are used to produce an algorithm for automatic exploration in Section 5. This algorithm is evaluated for both static schedule height reduction and runtime performance in Section 6. Finally, Section 7 concludes.

2. Related Work

Many people have have recognized the promise of automatic compiler retargeting, either by directly retargeting the compiler or through the automated synthesis of a compiler. Collberg describes how to reverse-engineer an entire compiler by querying an existing C compiler for the machine being reverse-engineered [6]. In a related area, Engler et al. describe a technique for reverse-engineering the operation encodings for a machine via queries to an existing assembler [12]. Yotov et al. describe how to use microbenchmarks and an existing C compiler to automatically derive microarchitectural hardware parameters related to memory, such as cache size and associativity [35]. Dupré et al. describe a method to automatically derive an instruction scheduler for machines given a cycle-accurate VLIW simulator with superscalar-like dynamic reordering capabilities that can report the status of each instruction during execution [9]. Description languages, such as ADLs, and associated tools for automating the synthesis of compilers and simulators from a single description have been proposed [32, 37, 37, 23].

Non-automatic retargeting of compilers has also been studied. Bradlee et al. created Marion, a system for retargeting instruction selection, instruction scheduling, and register allocation from a machine description [3]. Hanono et al. describe similar retargeting in the embedded processor domain using the AVIV retargetable code generator [18].

Researchers have also explored the area of representations for efficient hazard scheduling. Resource maps were proposed by Davidson et al. [7]. Additional representations based on finite state machine hazard detectors have also been proposed [2, 30]. Work also exists on factoring and optimizing resource-map-based machine descriptions to improve compile time [10, 17].

While instruction schedulers can be automatically retargeted, no technique yet proposed can do so automatically from a black-box machine. This paper draws inspiration from Baker, who advocates the automatic reverse-engineering approach to retargeting of instruction schedulers [1].

3. Structural Hazard Discovery Problem

The task of the hazard detector is to determine whether an instruction schedule contains structural hazards. On an idealized machine or machine model[1], structural hazards are revealed when a schedule takes longer to execute than expected. Thus, the target machine itself can be used as a hazard detector during scheduling [1]. Unfortunately, this is impractical as instruction schedulers query the hazard detector repeatedly during the scheduling of each instruction. This means that adding even a small amount of overhead to each query will quickly accumulate, significantly increasing compilation time. Additionally, the overhead of each query is unlikely to be small, as it includes the time to prepare the code for execution (e.g. register allocation), to execute it, and to evaluate the result.

Since querying the actual machine during compilation is expensive, the alternative is to query the machine *a priori*, that is, before the compiler is created, and to store the results in a *conflict database*. Ideally, the conflict database is both sound, all structural hazards it contains are correct, and complete, it contains all structural hazards. One methodology for obtaining this is to reverse-engineer the machine or machine model, as the machine is both sound and complete with respect to the structural hazards that exist in it. Unfortunately, as we now prove, an algorithm building a conflict database solely by reverse-engineering cannot know when the conflict database is complete. Despite this theoretical impossibility, the remainder of this paper shows that reverse-engineering in practice can identify enough (possibly all, but not knowingly so) structural hazards to produce high-quality instruction schedules at compile time.

The theoretical impossibility for perfect structural hazard determination stems from the unbounded number of possible instruction schedules with hazards. Since exhaustive exploration of all schedules is impossible, some finite set must be used instead. Unfortunately, as we will prove, all finite subsets can miss some structural hazards. Conceptually, there may exist some structural hazard which manifests itself only for schedules larger than any in the fi-

[1] An idealized machine is one free from noise generated by cache effects, page faults, etc. Section 5.2 discusses how to measure schedules on real machines.

nite set. Before formally proving this, we will introduce some terminology to aid in discussion.

Definition 1. *An* instruction schedule*, or* schedule*, is a collection of (instruction, issue time, issue slot) tuples. Each tuple identifies an instruction that is to be executed, the time that it should be issued, and the issue slot, within that cycle, in which it is issued.*

Definition 2. *A* static schedule *is a schedule that describes the compiler's belief about how the instructions will execute.*

Definition 3. *An* execution schedule *for a given static schedule and machine is the schedule of when instructions are* actually *issued on the given machine, instead of the time that the compiler believed the instructions would issue.*

Definition 4. *The* schedule height *of a schedule is* $(\textit{last instruction issue time} - \textit{earliest instruction issue time} + 1)$.

Definition 5. *A* machine *accepts a static schedule and outputs the schedule height of the execution schedule.*

Definition 6. *A* resource map *is a set of resource usage tuples (R, t), where R is the resource used and t is the time the resource is used, relative to the issue time of the instruction.*

Definition 7. *An instruction with* scheduling alternatives *has a set of resource maps, each of which describes a potential resource usage when it executes. Scheduling alternatives that allow an instruction to execute without a structural hazard are chosen before ones that will cause a structural hazard on an ideal machine.*

Using the definitions above, we now formally define the *Structural Hazard Discovery Problem* (SHDP). The problem is to create an algorithm A that can create another algorithm that can perfectly answer any and every query about structural hazards for a machine M. That is, A constructs another algorithm A_M, a generalization of the conflict database to an algorithm. A_M determines if a given static schedule contains a structural hazard on M. A can query M with a finite, though unbounded number of static schedules. To reflect that the machine M can only be queried *a priori*, A_M is not allowed to query M. We prove that this is impossible for machines that can be characterized by resource maps, which implies that it is impossible in the general case. Resource maps can be used to describe, among others, modern processors and machines with irregular constraints [31], as are often found in embedded processors. Theroem 1 proves that SHDP is not solvable for a general class of machines.

Theorem 1. *SHDP for machines whose structural hazards can be characterized by resource maps is unsolvable.*

Proof. We will prove the theorem by contradiction. Assume there is a deterministic algorithm A that, given a machine M (that can be described using resources), can produce, by querying M with a finite number of schedules, an algorithm A_M. A_M can determine if an input static schedule contains a structural hazard. Let C be the set of all static schedules tested by A on M. Let h be the maximum static schedule height in C. Finally, let t be the maximum time that a resource is used in any schedule in C.

Construct a new machine M' from M, where M' has the same resources and resource usage patterns as M with the following additions. Let S represent the maximum number of instructions in any tested schedule, and $h' = max(h, t) + 1$. Let M' have S new resources, $R_1, R_2, \ldots, R_S$. For each instruction, I, in M with scheduling alternatives $a_1, \ldots, a_n$, let the same instruction in machine M' have $S \times n$ scheduling alternatives, $a'_{11}, a'_{12}, \ldots, a'_{nS}$. For each $1 \leq k \leq S$ alternative a'_{ik} uses resource R_k from time 1 to time h', relative to the instruction issue time. Figure 2 shows a resource assignment for $n = 1$ and $S = 3$.

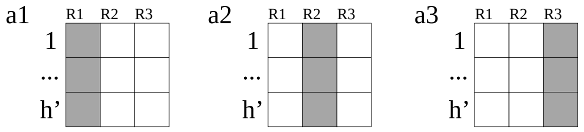

Figure 2. Example: $n = 1, S = 3$

Since the behavior of M is identical to the behavior of M' for all schedules whose height is $\leq h$, $A_M = A_{M'}$. For any instruction I, consider the schedule $((I, 1, 0), (I, 2, 0), \ldots, (I, S, 0), (I, S + 1, 0))$, which causes a conflict at time h' on M', but cannot cause a conflict in M by construction. Therefore, $A_M = A_{M'}$ will either accurately characterize machine M or machine M', but not both. This contradicts the assumption that A solves SHDP. □

Corollary 1. *The SHDP is unsolvable in general.*

Proof. It follows directly from Theorem 1 that SHDP is unsolvable in general because it is unsolvable for the special case of machines that can be described using resources. □

Since SHDP is unsolvable, it is impossible for an algorithm to provably determine all structural hazards of every machine *a priori*. Thus, it is also impossible for such an algorithm to knowingly create a complete conflict database. In practice, though, it is possible to bound the number of instruction schedules necessary to explore using machine properties such as the number of transistors, the maximum number of cycles a resource is used by a single instruction, the number of instructions in the ISA, etc.

Unfortunately, as shown in Section 4, even with this information, the space of instruction schedules that must be explored to discover all structural hazards typically remains too large to be exhaustively explored. However, it is possible to construct a good approximate conflict database in practice by identifying certain structural hazards and inferring the existence of additional structural hazards. The remainder of the paper describes the observations used to infer additional structural hazards, proposes an algorithm to construct the conflict database, and shows that it works well in practice.

4. Reducing the Instruction Schedule Space

As we saw in Section 3, building a complete conflict database would require testing an infinite number of schedules. To allow reverse-engineering to finish in a finite amount of time, we forgo building a complete conflict database by searching only a finite subspace of instruction schedules. The following subsections show how to select a candidate finite subspace that can be explored in a tractable amount of time. Section 4.1 selects an intractably large, but finite subspace, while the remaining subsections prune this subspace down to a tractable size, which can be searched in a few hours.

4.1 Pipelining

If one can assume that there exists a bound on the length of time an instruction can have an effect on the timing of other instructions, then only schedules whose height is less than or equal to the bound need be considered. As an approximation, only instruction schedules that are within a time bound are chosen for exploration. Using this, the size of the schedule space is I^B, where I is the number of instructions in the ISA and $B = w \times d$, where d is the time bound and w is the issue width of the machine.

A first approximation of d can limit it to the depth of the pipeline. However, for modern wide-issue processors, d can be further approximated to 1. This observation comes about because same-cycle hazards are usually more prevalent than inter-cycle hazards. Instruction interaction across cycle boundaries usually occurs because a resource is consumed for several cycles. Since modern and embedded processors attempt to fully pipeline functional units, conflicts arising from multi-cycle resource usage are diminishing. Thus, for many processors, the set of instruction schedules can be limited in size to the width of the machine, essentially ignoring depth. For processors that are not fully pipelined, it is typically possible to limit d to small integers.

The *No Depth* bar in Figure 3 shows the results of only detecting structural hazards in the current cycle, equivalent to setting $d = 1$. The x-axis shows the results for five machines across a set of benchmarks, details of which are in Section 6. The y-axis is the speedup achieved over scheduling without hazard detection. The *Full* bar denotes speedup from using resource maps, while the *No Depth* bars shows the speedup obtained with hazard detection limited to instructions in the current cycle. As the graph shows, the depth of the instruction schedules is only relevant on the SPARC machine.

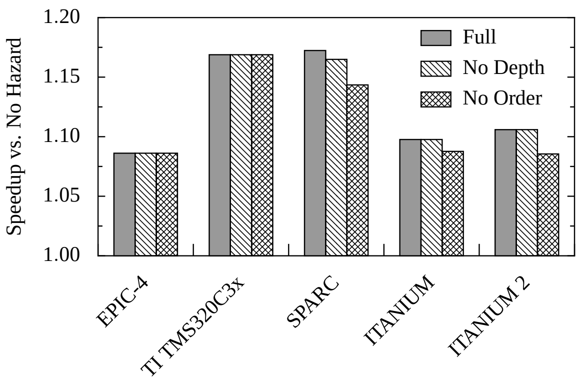

Figure 3. Speedup over scheduling without resource maps for scheduling with full resource maps, single-cycle resource maps, and single-cycle resource maps without order.

To illustrate how large and time-consuming the schedule space can be to exhaustively explore, consider the largest and most complex machine described in Section 6.1, the Itanium 2, which has approximately 300 instructions and can issue 6 instructions per cycle. Assuming these values were all known beforehand, an exhaustive exploration must still search $300^6 = 7.29 \times 10^{15}$ instruction schedules. Assuming a testing implementation that can test 10 schedules each second, it would take $2,311,643$ years to explore all these schedules.

4.2 Ordering

Another reason for the large instruction schedule space is that every possible order of each combination of instructions is considered independently. In practice, however, the order of instructions issued in the same cycle tends not to matter. This occurs because instructions tend to utilize the same resources or an identically behaving set of resources when their position within a cycle is changed.

To take advantage of this, the set of instruction schedules explored in the reduced space is limited to all unique multi-sets of instructions. By exploring only the combinations of instructions using representative orders, the size of the search space is reduced from $I^{w \times d}$ to $\left(\begin{array}{c} I+w-1 \\ w \end{array}\right)^d$ [24], assuming empty slots are filled with *nops*. *nops* are instructions that utilize a fetch resource, but consume no back-end resources. When a structural hazard is found in the reduced space, it can be mapped back to the original space by taking all permutations of the multi-set.

While the ordering among instructions in the same cycle may be considered irrelevant, a specific ordering is still chosen for testing. While a random ordering could be used if ordering is indeed irrelevant, we have empirically observed that previously tested orders provide better performance since their behavior is known. Though ignoring the ordering of instructions is made easier by limiting the instruction schedules to depth 1, the notion of order independence can be extended to multi-cycle situations.

The *No Order* bar in Figure 3 shows the results of only detecting structural hazards in the current cycle and ignoring the order of instructions when doing so. While ignoring the order of instructions does matter in terms of speedup, the loss is not great, meaning that effective scheduling can still be performed without knowledge of order-based structural hazards. Moving to order ignorant instruction schedules, the Itanium 2 search space is reduced to $\left(\begin{array}{c} 305 \\ 6 \end{array}\right)^1 = 1.06 \times 10^{12}$ instruction schedules, which can be searched in $33,742$ years (again assuming 10 schedules per second).

4.3 Categorization

Another observation that can be used to reduce the space is that processors' instructions tend to fall into one of several categories with respect to their resource usage. These categories may align with the categories of functionality, such as Load, Store, ALU, and Branch, but often partitioning solely by functionality is not sufficient and, if set incorrectly *a priori*, can even be detrimental. Consequently, in the algorithm presented in this paper, categories are discovered automatically.

We can define a category of instructions as a set of instructions exhibiting the same behavior in schedules. This allows a canonical instruction to be chosen from each category and be used to represent all instructions in the category. The set of conflicts determined for the set of canonical instructions effectively represents the conflicts of the entire instruction space. By using canonical instructions, the size of the I term in I^B is greatly reduced. In practice, this number often reduces I from several hundred instructions to around a dozen or so canonical instructions.

Each instruction category can be viewed as an equivalence class of instructions. Any pair of schedules, S_1 and S_2, are equivalent **iff** for any triple $(i, t, s) \in S_1$ there is a matching tuple $(i', t, s) \in S_2$ such that i and i' are in the same category. A mechanism to automatically categorize instructions is presented in Section 5. The fully reduced search space for the Itanium 2, which contains approximately 2 dozen categories, now has $\left(\begin{array}{c} 29 \\ 6 \end{array}\right) = 4.7 \times 10^5$ instruction schedules, which can be searched in 13.2 hours (once again assuming a rate of 10 schedules per second, pipelining, and no order). Without the order assumption, there would be $24^6 = 1.9 \times 10^9$ instruction schedules that would take 6 years to search.

5. Building the Conflict Database

In this section, the observations from Section 4 are used to develop an algorithm for building the conflict database. The high-level algorithm for building the conflict database is shown in Figure 4.

The set C represents the set of all categories and all instructions start out in a single category. A schedule of randomly chosen instructions is created to initialize a random walk. The random walk continues until it finds a category suitable for splitting, returning the category to split, Q, and the schedule template, T, with which to split it. The random walk and category splitting are described in Section 5.1. $Split(Q, T)$ splits the instructions in Q

```
1:  $C$ = {{All Instructions}}
2:  $Conflict\_Database$ = {}
3:  **while** TRUE **do**
4:      $Q, T$ = Random_walk()
5:      $C = (C \setminus Q) \cup Split(Q, T)$
6:      $S$ = set of schedules formed from $Canonical(C)$
7:      **for** $s \in S$ **do**
8:          Query the machine with static schedule $s$
9:          **if** $s$ contained a structural hazard **then**
10:             Update($Conflict\_Database$, $\{s\}$)
```

Figure 4. High-level algorithm for building the conflict database

into a set of new categories $C_1, C_2, \ldots, C_n$. After this, a canonical instruction is chosen from each category, enforcing that the $Canonical(C_i) = Canonical(Q)$ if $Canonical(Q) \in C_i$. The set of schedules formed by the canonical instructions is exhaustively searched. Since $Canonical(Q)$ is still a canonical instruction for some category, the results of all previously tested instruction schedules are still valid. Thus, nothing needs to be removed from the conflict database and the results of this search need only be used to add new conflicts to the database. The algorithm then repeats by starting a new random walk.

The output of this algorithm is a categorization and a conflict database, which is valid after any exhaustive search on the canonical instructions has finished. Any of the conflict databases and categorizations thus produced can be used for hazard detection, though the algorithm creates a more accurate conflict database as it is given more time and finds more categories. To use the conflict database and categorization during scheduling, the instruction schedule is converted into a canonical, depth-ignorant, and order-ignorant version. This version is then used to query the conflict database and return whether a structural hazard exists. We now describe the process of automatically classifying instructions into categories (steps 4 and 5, Section 5.1) and methods for querying actual hardware (step 8, Section 5.2).

5.1 Instruction Categorization

Though perfect formation of instruction categories requires knowledge of all structural hazards, in practice, categories can be approximated by observing differences in schedule execution times. If two schedules (with no data dependencies) differ in only one instruction and have differing execution times, then they experience different structural hazards. Since the schedules differ only by a single instruction, the resource utilization, and therefore the category, of the two instructions must also differ.

Given an initial categorization of instructions, the above observation is used to refine the categories. Any pair of schedules with the properties described above can be used to partition a single category C, which occurs as follows. A *schedule template*, T, is formed by taking the common portions of the schedule pair and leaving the differing portion vacant. Since the pair differs by one instruction, there is one slot in the resulting template where each member of the pair of schedules had a different instruction. The original category C is discarded, and the instructions in C are placed into new categories, $C_1, C_2, \ldots$, based on the execution times observed when inserting each instruction from C into the empty slot in T. That is, all of the instructions in C_n have the same execution time when placed into T.

An obvious and simple initial categorization is to make all instructions a single category. If schedule pairs with the appropriate properties can be found, the refinement process can build a more accurate categorization. To solve the problem of finding appropriate schedule pairs, a random walk through the space of instruction schedules is performed.

The random walk starts with a seed schedule, chosen at random from the set of all instructions I. The walk then randomly chooses one instruction in the schedule and replaces it with a random instruction chosen from I. Information is gleaned with every iteration because the new schedule and old schedule differ by exactly one instruction. Additionally, the random walk is very simple, requiring no complicated heuristics or fore-knowledge, meaning that progress can be made quickly without having any assumptions about the underlying hardware. Though a random walk is not the only means by which to categorize instructions, as Section 6 will show, the random walk is a highly effective means for forming categories even when the categorization process ignores order and schedule depth.

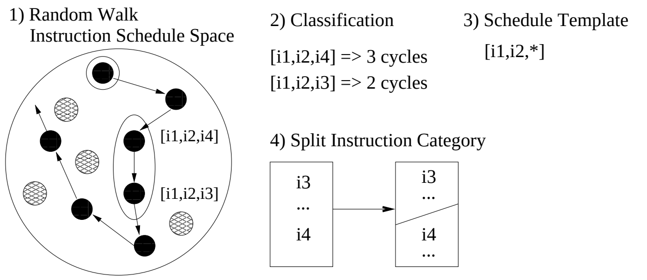

Figure 5. Example of classification

The instruction categorization process is illustrated in Figure 5. Each step in the walk randomly changes one instruction from the previous schedule. When two adjacent schedules in the walk are found with different execution times, a schedule pair has been found. For example, in Figure 5, the random walk steps from the schedule $[i1, i2, i4]$ to $[i1, i2, i3]$. Since $i3$ and $i4$ belong to the same category and have different execution times, the category is split. Each instruction from the category is placed into the last slot of the schedule, with each unique execution time forming a new category.

5.2 Querying Actual Hardware

If the machine model queried is actual hardware, detection of resource over-subscription may require some engineering effort. A naïve approach would simply present the hardware with a schedule and measure the cycle count returned by the processor. If the cycle count is greater than expected, a conflict has occurred. Unfortunately, real processors may not have the measurement accuracy to perform the test, as most processors do not have cycle-accurate performance counters. Others, such as the Itanium 2, have performance counters which are accurate to a certain tolerance. If the inaccuracies can be modeled as zero-mean, such as from cache misses, page faults, etc., or transient noise, then the schedule can be executed in a loop to negate the noise generated. Existing research demonstrates strategies to determine execution times of pipelined processors with non-zero mean noise [11, 29].

When executing instructions in a loop to cancel out noise, care must be taken to avoid conditions in the processor that are not the result of structural hazards. Specifically, each iteration of the loop should effectively clear the pipeline so that there are no effects that build up in the loop and cause a stall that executing the sequence only once would not.

In general, such conditions come from instructions that access memory or instructions that utilize a resource for a long period of time. Each instruction that accesses memory should utilize the same address each time around the loop, but should not have the same or overlapping address as another memory instruction. This avoids output, anti, and flow dependences through memory and avoids the variable latency of memory instructions by always hitting in the cache. To avoid stalls that build up from resources used for several cycles, the loop should be buffered with *nops* sufficient to flush the pipeline. Note that the number of *nops* needed can be an overestimation, and that the system itself can explore to find an approximate number of cycles of *nops* needed to effectively empty the pipeline. Additionally, care must be taken to avoid corner cases that can arise from misalignment of the loop itself to unintended cache stalls because of the address chosen [5]. Automatic determination of cache behavior is beyond the scope of this work, but has been explored by Yotov et al. [35]. In this paper, all such conflicts were avoided by ensuring that memory addresses assigned to each memory reference were independent of such effects.

Additionally, depending on the type of ISA, the algorithm may be unable to present to the hardware an instruction schedule. Instead, it may only be possible to present a sequence of instructions. If the schedule height of all queries is limited to 1, then all instruction schedules correspond directly to instruction sequences. If it is greater than 1, then data dependences can be used to force instructions to execute according to a particular schedule. This is not necessary for the machines explored in this work, and the presentation of this extension to the technique is beyond the scope of this paper.

Another issue with hardware queries arises when a machine fails silently upon structural hazards violation. In such a case, a compiler can produce invalid, not just inefficient, schedules. To support this type of machine, the querying framework needs to be expanded to check results of the computation. Note that the technique would still be machine independent; the instructions and their semantics are still a given. This paper does not addresses this because the machines of interest do not fail silently with structural hazards; they either stall or throw an exception.

6. Evaluation

Section 5 presented an algorithm that does not build a perfect conflict database. As a result, there are two potential pitfalls in the algorithm presented. First, the algorithm could produce a conflict database that sacrifices too much performance. This inaccuracy can come about because the machine model queried does not match the actual hardware or from the approximations made during the conflict database formation. Second, the algorithm could take too much time to produce a useful conflict database. This section shows that neither occurs in practice for a variety of machines.

6.1 Exploration Setup

The conflict database was implemented as a simple hash table and contained only the structural hazards found by the algorithm in Section 5. To determine if an instruction schedule contains a structural hazard, the instruction schedule is converted to a canonical, depth-ignorant, and order-ignorant version and then used as a query to the conflict database. This allowed efficient lookup for the scheduler while maintaining a reasonable conflict database size. Implementations based on other data structures are possible, as nothing in the technique assumes a specific implementation. To determine the quality of the algorithm, it was evaluated on several machines.

Idealized EPIC machine A 4-issue machine with 2 ALUs, 2 floating point (FP) ALUs, 1 Divide unit (DIV), 2 memory units, and 1 branch unit. There are no ordering or depth constraints in this processor.

TI TMS320C3x A 2-issue machine with resources for auxiliary register file access, 1 ALU, 1 multiply unit (MUL), 2 memory ports, and 1 branch per cycle. All resources modeled for the TI TMS320C3x have no ordering or depth constraints.

SPARC Viking 8 A 3-issue machine with resources for 1 FP ALU, 1 FP MUL, 1 FP DIV, 1 Shifter, 4 Register File (RF) Read Ports, and 2 RF Write Ports. There are several instructions in the Viking 8 (e.g. branches) that utilize resources for 2 or more cycles. Additionally, branches must be the last instruction in a set of issuing instructions.

Itanium (Merced) The Itanium is the first implementation of the Intel Itanium Processor family and is a six-issue machine. It has 2 integer units, 2 FP units, 2 memory units, and 3 branch units. There are no instructions that utilize a resource for more than one cycle; however, certain instructions have ordering constraints. Additionally, the execution units are not symmetric with respect to the instructions they can execute (e.g. certain instructions can only execute on one functional unit even though there are two units in the class).

Itanium 2 (McKinley) The Itanium 2 is the second implementation of the Intel Itanium Processor Family (IPF) and is a six-issue machine. It has 2 integer units, 2 FP units, 4 memory units, and 3 branch units. There are no instructions that utilize a resource for more than one cycle; however, certain instructions have ordering constraints. Additionally, the integer and memory unit sets are not symmetric with respect to the instructions they can execute.

To evaluate the effectiveness of a conflict database for hazard detection, several detectors were implemented for comparison.

No Hazard The base hazard detector for comparison is a hazard detector that never detects a hazard.

Full Hazard A hazard detector that uses existing complete resource maps to determine if a structural hazard exists.

No Depth A modified *Full Hazard* detector that uses resource maps with structural hazard detection limited to the current cycle, which ensures that the only structural hazards found are intra-cycle structural hazards.

No Order This detector is the same as the *No Depth* detector except that the instructions are arranged in an arbitrary order in the current cycle before hazard detection is performed. The order chosen is a sorted order based on a random numbering of the instructions, which ensures an arbitrary but consistent ordering. This detector effectively ensures that the only structural hazards that occur are ones that occur regardless of order in the current cycle.

Conflict Database A hazard detector based on an automatically generated conflict database for a randomly selected set of canonical instructions. To determine if an instruction schedule contains a structural hazard, the instruction schedule is converted to a canonical, depth-ignorant, and order-ignorant version and then used as a query to the conflict database. The conflict database is then queried with the canonical schedule.

The *No Hazard* and *Full Hazard* detectors represent the lower and upper bounds, respectively, of the performance of the *Conflict Database* hazard detector on an optimal scheduler. The hazard detectors were developed in the OpenIMPACT compiler [28]. They were evaluated by scheduling the SPEC CINT2000 benchmark suite, compiled with standard optimizations using the OpenIMPACT compiler. The OpenIMPACT compiler uses a list-scheduling algorithm with a heuristic that gives higher priority to instruction that lead to exits. Profiles were gathered on train inputs and runtime performance was measured on reference inputs.

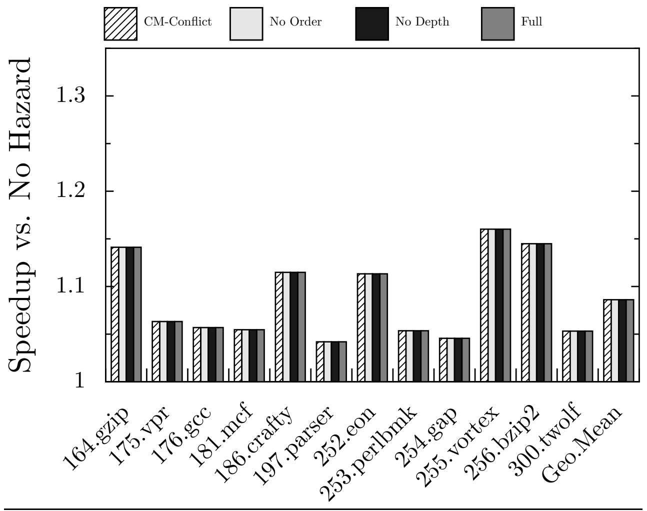

Figure 6. Static Schedule Height - EPIC-4

6.2 Reverse-Engineering Using a Compiler

To determine the effectiveness of the technique described in Section 5, we first run the technique on a compiler that models the resource usage during scheduling.

The conflict database is formed by querying the schedule manager (SM) in the OpenIMPACT compiler. SM has a complete resource map for each of the machines and can be queried with a static schedule. An in-order version of SM that schedules without reordering instructions, but that respects all structural hazards in the hardware, was developed to get the true static schedule height. Each benchmark was scheduled using each hazard detector and then a total static schedule height was calculated by multiplying the profile weight by the in-order schedule height.

Figures 6-8 show the speedup of each hazard detection mechanism over the *No Hazard* detector for three of the machines. The *Full Hazard* detector provides a theoretical upper bound on the speedup that can be obtained by any of the detectors as this detector is aware of *all* conflicts. This bound is theoretical due to the heuristic nature of the list scheduler which may produce a better schedule with different or less accurate information. The *No Depth* and *No Order* hazard detectors provide theoretical upper bounds on the speedup that the categorization and exploration algorithm can obtain, as discussed in Section 4. The conflict database formed from the compiler is the *CM-Conflict* bar.

Figure 6 shows the results for the generalized EPIC processor, which represents a broad class of processors (such as the FR500 [33] and the Lx/ST200 [14]) that are relatively wide and that have sufficient structural hazards to warrant scheduling with hazard detection. Since the machine has no depth or ordering constraints, the technique finds all categories and achieves performance equivalent to using full resource maps. While full resource maps for this machine are relatively simple, the purpose of this machine is to show that the technique can perform well in general. The other machines show how the technique performs in the presence of more complex structural hazards.

Figure 7 shows the results for the TI TMS320C3x. The graph shows that the conflict database hazard detector performs the same as the *No Order*, *No Depth*, and *Full* hazard detectors, meaning that the categorization algorithm found all categories and that the conflict database represented all conflicts. This is not surprising given that the resources for the TI TMS320C3x have no ordering or depth constraints.

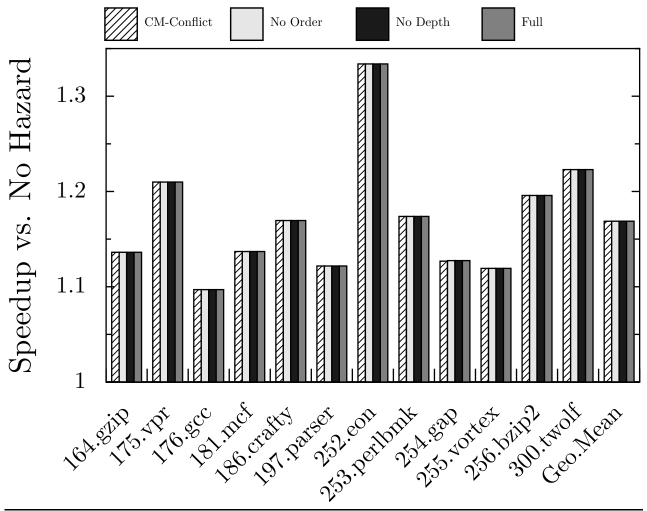

Figure 7. Static Schedule Height - TI TMS320C3x

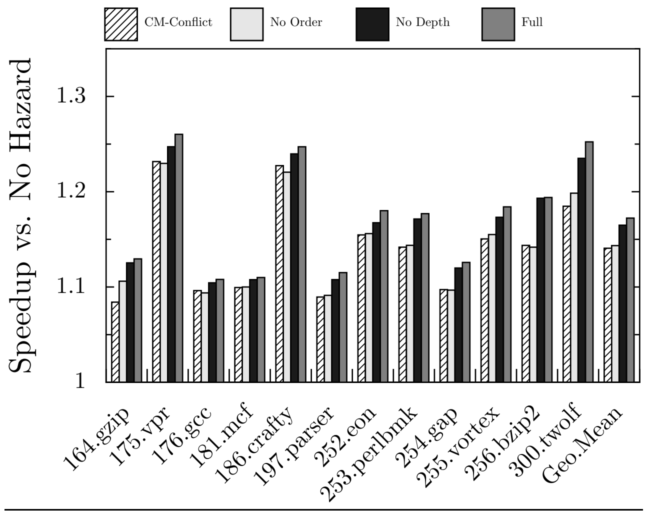

Figure 8. Static Schedule Height - SPARC Viking 8

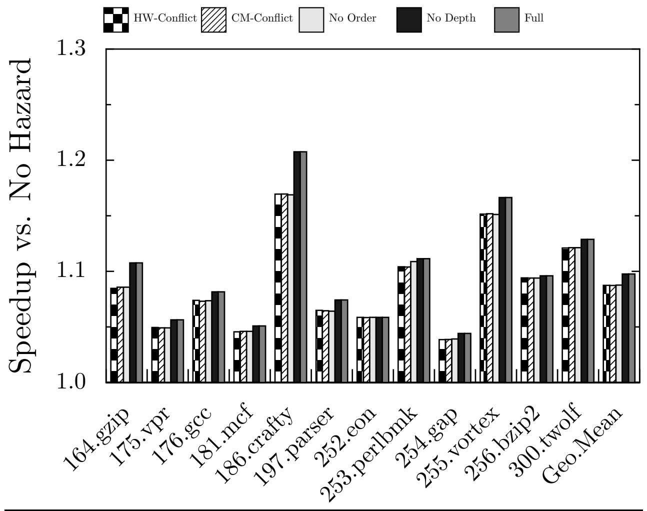

Figure 9. Static Schedule Height - Itanium

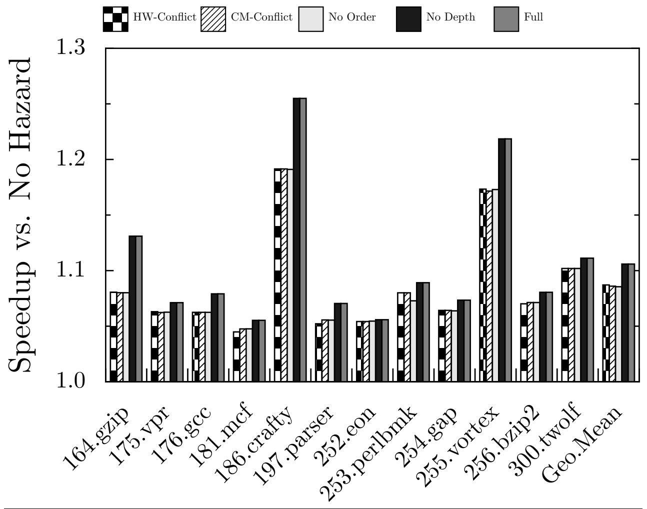

Figure 10. Static Schedule Height - Itanium 2

Figure 8 shows the results for the SPARC Viking 8 processor. Since the SPARC contains instructions with depth and order constraints, particularly branches, there is a performance loss when ignoring order and depth. As the figure shows, the Conflict Database hazard detector is able to obtain 94% of the potential speedup, and performs as well as the *No Order* hazard detector. This implies that most of the relevant categories and thus conflicts were found during the conflict database formation.

As Figures 9 and 10 show, the *CM-Conflict* conflict database realizes 89% and 81% of the potential performance improvement for the Itanium and Itanium 2, respectively. The graphs also show that the conflict database is as good as resource maps for hazard detection provided instruction order is ignored. Further analysis of the produced schedules indicate that the `Deposit` and `Extract` instructions (which must be the first integer instruction issued in a given cycle) account for the majority of the slowdown due to the *No Order* heuristic. This suggests that one could explore methodologies for determining if an instruction is an order-dependent instruction to recover this lost performance.

Note that the hazard detectors that use the *CM-Conflict* conflict database (which ignores hazards due to instruction order) slightly outperform the *No Order* detector on several benchmarks for the SPARC machine and 253.perlbmk for the Itanium 2 machine. This occurs because the order of the instructions used to detect hazards is different from the scheduled order. Specifically, the order-ignorant resource maps encode a conflict for a sequence of instructions if *even one* ordering of that sequence conflicts. This occurs even if the actual instruction order in the final schedule has no conflict. The generated hazard detectors, however, fail to detect that a conflict exists at all, generating a better schedule.

We now examine the time required to derive the instruction categories and exhaustively search the reduced space. To form the *CM-Conflict* conflict database, the algorithm presented in Section 5 was run on a 2.8GHz Pentium 4 with 1 Gb of memory and averaged approximately 15,000 instruction schedule tests per hour or 4 tests per second. The implementation of the random walk used in this evaluation is itself subject to the no-order and no-depth constraints. Because of this, it may not find all the categories that actually exist. Additionally, due to the heuristic nature of the list scheduling algorithm used in IMPACT, more accurate structural hazards may cause a less optimal schedule to be chosen.

Figure 11 shows the speedup of each conflict database formed after each exhaustive search for 4 of the machines. The EPIC-4 conflict database converged to the resource map speedup in under 10 minutes and is not shown. For the remaining machines, the top of the graph represents the speedup of the *Full* hazard detector as shown by the *Geo.Mean* bar in Figures 7-10. As Figure 11 shows, the algorithm quickly achieves a large portion of the total conflict database speedup within 10 minutes. After 2 hours, it has effectively determined all conflicts that it is likely to find that are relevant to schedule performance. After halting the algorithm, it has achieved, 100%, 82%, 89%, and 81% of the *Full* hazard detector performance for the TI, SPARC, Itanium, and Itanium 2 machines respectively.

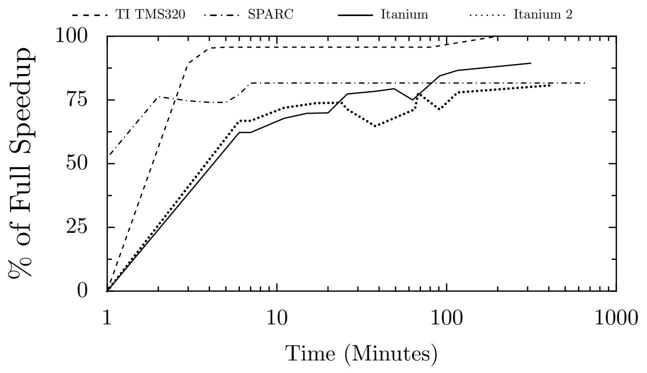

Figure 11. Schedule Height Speedup over time for *CM-Conflict* as a percentage attained by *Full*. Lines end when search is terminated.

6.3 Reverse-Engineering Using the Hardware

The previous section dealt with reverse-engineering the structural hazards from a compiler. A more realistic scenario might involve directly querying the hardware to generate the conflict database. This section explores the ability to query to machines and shows that the generated conflict database performs as good as the conflict database generated from a compiler.

6.3.1 Hardware Conflict Database Schedule Height

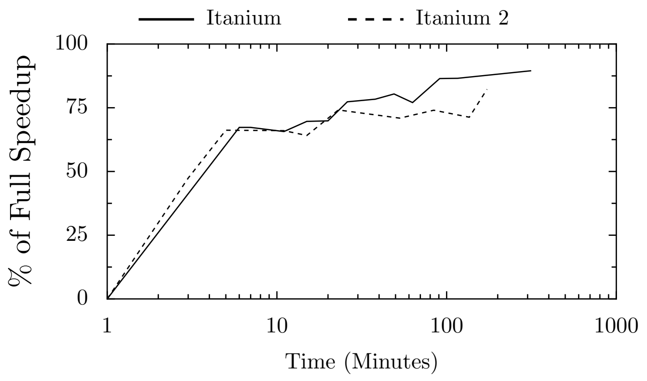

Figure 12. Schedule Height Speedup over time for *HW-Conflict* as a percentage attained by *Full*. Lines end when search is terminated.

The hardware queried conflict database was formed for both the Itanium and Itanium 2. Queries to the hardware were performed by placing the instruction schedule to be tested inside a counted loop that was executed 100,000 times. This number was arrived at by hand testing a small set of schedules until the one-time costs of the loop, as discussed in Section 5.2, were removed when dividing by the iteration count. The loop was also buffered with 6 cycles of nops to ensure the pipeline was empty and to avoid structural hazard across loop iterations. Execution times were obtained using the performance counters of the IPF architecture and the *pfmon* tool [13]. The Itanium machine used was a HP i2000 workstation with a 733Mhz Itanium and 1Gb of memory, while the Itanium 2 machine used was a HP workstation zx2000 with a 900Mhz Intel Itanium 2 processor and 2Gb of memory. Both systems run Redhat Advanced Workstation 2.1. The system averaged 12,000 instruction schedule tests per hour or 3 tests per second.

The *HW-Conflict* bar in Figures 9 and 10 represents the speedup of the hardware queried conflict database over the *No Hazard* hazard detector and shows that it realized 89% and 81% of the potential performance improvement for the Itanium and Itanium 2 machines, respectively. Essentially, the *HW-Conflict* conflict database achieves equivalent speedup to the *CM-Conflict* conflict database generated from the compiler. This shows that hardware can be used to build an effective conflict database.

Figure 12 shows static schedule height speedup over time for the *HW-Conflict* conflict database after each exhaustive search on the Itanium and Itanium 2 machines. Once again, the top of the graph represents the final speedup of the *Full* hazard detector. As with the *CM-Conflict* hazard detector, the algorithm still achieves a large portion of the total conflict database speedup within 10 minutes. Within 3 hours, it has effectively determined all conflicts that are relevant to schedule performance, given the no-order, no-depth assumptions. When the algorithm is halted, the *HW-Conflict* hazard detector has achieved 89% and 81% of the *Full* hazard detector performance for the Itanium and Itanium 2 respectively.

6.3.2 Hardware Conflict Database Runtime Performance

Since the machines being used are real hardware, it is also possible to generate runtime numbers for the hazard detectors. Figures 13 and 14 show that the execution time speedups for Itanium and Itanium 2 processor, respectively. As the graphs show, static schedule height improvement does not translate directly into runtime performance improvement. Overall, full resource maps lead to a 2.70% and a 3.22% gain in runtime performance on the Itanium and Itanium 2 processor, respectively. For the Itanium, the *HW-Conflict* conflict database realizes 88% of the possible performance gain, while the *CM-Conflict* conflict database achieves 84% of the possible performance gain. For the Itanium 2, the *HW-Conflict* conflict database achieves realizes 84% of the possible performance gain, while the *CM-Conflict* conflict database realizes 82% of the possible performance gain.

Since schedule height does not directly translate into runtime performance, the overall effect of hazard detection is reduced on real hardware. This is due to differences in input sets which affect the accuracy of the scheduling heuristic, variable latency instructions, the heuristic nature of the scheduling algorithm, and different register pressure from different schedules.

Figure 15 shows the runtime speedup over time for the *HW-Conflict* conflict database after each exhaustive search on the Itanium and Itanium 2 machines. For each machine, the top of the graph represents the final speedup of the *Full* hazard detector as shown by the *Geo.Mean* bar in Figures 13 and 14. As Figure 15 shows, the algorithm quickly achieves a large portion of the total conflict database speedup within 10 minutes. However, due to the variability of the dynamic runtime, the speedup over time is harder to discern. As the lines in the Figures show, the *HW-Conflict* actually achieves better performance earlier in the process, due to varying dynamic effects. Even so, within 3 hours, it has effectively determined all conflicts that are relevant to schedule performance. When the algorithm is halted, the *HW-Conflict* hazard detector has achieved 88% and 84% of runtime gain of the *Full* hazard detector performance for the Itanium and Itanium 2 respectively.

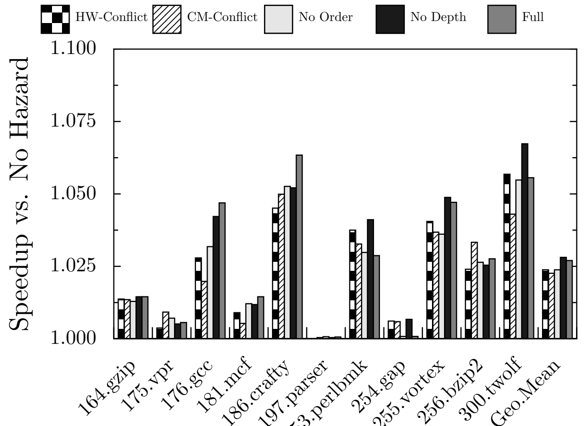

Figure 13. Runtime Speedup - Itanium

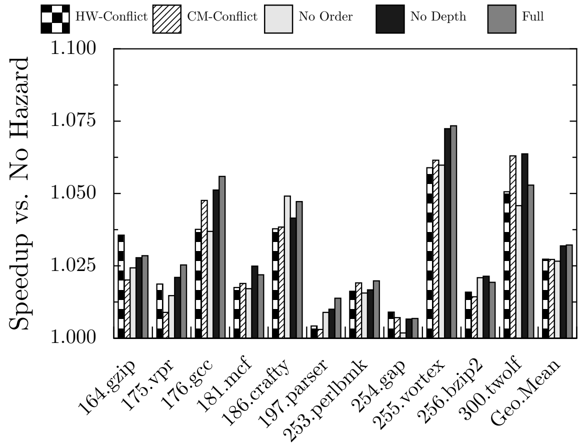

Figure 14. Runtime Speedup - Itanium 2

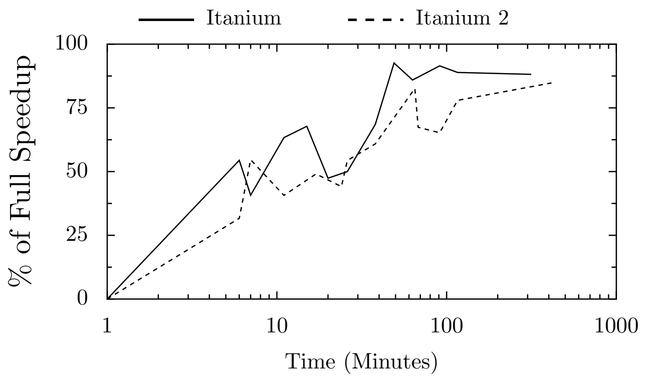

Figure 15. Runtime Speedup over time for *HW-Conflict* as a percentage attained by *Full*. Line ends when search was terminated.

6.4 Verifying Man-Made Machine Descriptions

As discussed in the introduction, man-made machine descriptions are tedious to create, which often leads to errors in the description. The Itanium 2 machine description was created in August 2000 and has been used for the IMPACT group's research into the Itanium architecture since that time. Given this, errors in the machine description are likely to have been found and corrected. However, the reverse-engineering described here was able to find errors in the resource maps for three instructions, shr, shl, and $shr.u$.

As a simple test of the validity of IMPACT's resource maps, we compared the number of times each instruction could issue with itself on both the manually-derived resource maps (SM) and the automatically reverse-engineered conflict database. This means that for a given instruction A, the schedules AA, AAA, and so on were tested. This allowed us to determine if any instruction had an inappropriate number of resources assigned to it in either case.

This process uncovered that the immediate version of the shl, $shl\ r_1 = r_2, imm_6$, for the Itanium and Itanium 2 were not properly modeled in the SM resource maps of the IMPACT compiler. Specifically, the immediate version of the shl is a pseudo-op of $dep.z$, which can only execute as the first integer type instruction in a cycle. Thus, only one immediate shl can execute per cycle. However, the register version of shl, $shl\ r_1 = r_2, r_3$, is not a pseudo-op and can issue on either integer unit, allowing up to two to execute per cycle. The IMPACT compiler's resource maps incorrectly indicate that the immediate version of shl uses the same resources as the register version. This error does not exist because resource usage is assigned by opcode, as IMPACT's register maps distinguish between different operand formats for each opcode. Instead, it is likely that the person creating the machine description did not read the Itanium 2 Instruction Reference Manual in conjunction with the Itanium 2 Processor Reference Manual. The instruction reference indicates that shl with an immediate is a pseudo-op of $dep.z$, but the processor reference says that $shl\ ar = ar, ar$ can issue twice in a cycle. ar is not explained and it is easy to mistake the resource usage for both versions of shl. The shr and $shr.u$ instructions have similar resource usage patterns and also suffer from this error.

7. Conclusion

This paper demonstrates that it is impossible to provably identify all the structural hazards in a machine ahead of time. It also presents a heuristic approach that reverse-engineers the structural hazards of the machine being explored using three observations that make reverse-engineer possible and tractable by selecting a finite subspace of the instruction schedule space to explore. Finally, it presents an algorithm to automatically determine instruction classifications that monotonically improves as categories are split.

We experimentally demonstrate that short exploration times can detect most structural hazards needed for scheduling. In fact, static schedules, constructed from exploring both the compiler and hardware for several machines, achieve 81-100% of the performance of scheduling with perfect structural hazard information. On two real IA-64 machines, these schedules achieve 82-88% of the runtime performance gain of perfect structural hazard information.

We plan to explore the effect of providing information to the algorithm, both for verification purposes and to assist in exploration. We are also interested in modifying the algorithm detect when an instruction is order-dependent or depth-dependent and to incorporate this into the conflict database. Finally, we would like to explore the practicality of our algorithm on machines with complex micro-architectural structures, such as a trace cache.

Acknowledgments

This work was inspired by conversations with Andrew Appel. Additionally, we would like to thank both the Liberty group and the IMPACT group, particularly John Sias, Ronald Barnes, and Sain-Zee Ueng. We also thank the anonymous reviewers for their insightful comments. This work has been supported by the National Science Foundation (NGS-0133712 and NGS-0305617) and Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the National Science Foundation or Intel Corporation.

References

[1] BAKER, H. G. Precise instruction scheduling without a precise machine model. *ACM SIGARCH Computer Architecture News 19*, 6 (1991).

[2] BALA, V., AND RUBIN, N. Efficient instruction scheduling using finite state automata. In *Proceedings of the 28th annual international symposium on Microarchitecture* (1995).

[3] BRADLEE, D. G., HENRY, R. R., AND EGGERS, S. J. The Marion system for retargetable instruction scheduling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1991), pp. 229–240.

[4] CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., WARTER, N. J., AND HWU, W. W. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture* (May 1991), pp. 266–275.

[5] COLLARD, J.-F., AND LAVERY, D. Optimizations to prevent cache penalties for the Intel Itanium 2 processor. In *Proceedings of the International Symposium on Code Generation and Optimization* (2003), IEEE Computer Society, pp. 105–114.

[6] COLLBERG, C. S. Automatic derivation of compiler machine descriptions. *Proceedings of the 2002 ACM Transactions on Programming Languages and Systems 24*, 4 (2002).

[7] DAVIDSON, E. S., SHAR, L. E., THOMAS, A. T., AND PATEL, J. H. Effective control for pipelined computers. In *Proceedings of the IEEE Spring Compcon 75* (February 1975), pp. 181–184.

[8] DEHNERT, J. C., AND TOWLE, R. A. Compiling for the Cydra 5. *The Journal of Supercomputing 7*, 1 (January 1993), 181–227.

[9] DUPRÉ, M., DRANCH, N., AND TEMAM, O. VHC: Quickly building an optimizer for complex embedded architectures. In *Proceedings of the International Symposium on Code Generation and Optimization* (2004), IEEE Computer Society.

[10] EICHENBERGER, A. E., AND DAVIDSON, E. S. A reduced multipipeline machine description that preserves scheduling constraints. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation* (May 1996), pp. 12–20.

[11] ENGBLOM, J. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Department of Information Technology, Uppsala University, Sweden, March 2002.

[12] ENGLER, D. R., AND HSIEH, W. C. Derive: A tool that automatically reverse engineers instruction encodings. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization* (2000).

[13] ERANIAN, S. Perfmon: Linux performance monitoring for IA-64. http://www.hpl.hp.com/research/linux/perfmon/, 2003.

[14] FARABOSCHI, P., BROWN, G., FISHER, J. A., DESOLI, G., AND HOMEWOOD, F. Lx: a technology platform for customizable vliw embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (2000), pp. 203–213.

[15] FISHER, J. A. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers C-30*, 7 (July 1981), 478–490.

[16] GRUN, P., HALAMBI, A., DUTT, N., AND NICOLAU, A. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. *IEEE Transactions on Very Large Scale Integration Systems 11*, 4 (2003), 731–737.

[17] GYLLENHAAL, J. C., HWU, W. W., AND RAU, B. R. Optimization of machine descriptions for efficient use. In *Proceedings of the 29th International Symposium on Microarchitecture* (December 1996), pp. 349–358.

[18] HANONO, S., AND DEVADAS, S. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *Proceedings of the 35th Design Automation Conference* (June 1998).

[19] HENNESSY, J. L., AND GROSS, T. Postpass code optimization of pipeline constraints. *ACM Trans. on Programming Languages and Systems 5* (July 1983), 422–448.

[20] IMPACT. Personal Communication, IMPACT Research Group, March 2004.

[21] INTEL CORPORATION. *Introduction to Microarchitectural Optimzation for Itanium 2 Processors: Reference Manual*. Santa Clara, CA, 2002.

[22] INTEL CORPORATION. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Santa Clara, CA, April 2003.

[23] KÄSTNER, D. TDL: A hardware description language for retargetable postpass optimizations and analyses. In *Proceedings of the Second International Conference on Generative Programming and Component Engineering* (2003), pp. 18–36.

[24] KNUTH, D. E. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[25] LAM, M. S. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1988), pp. 318–328.

[26] MILNER, C. W., AND DAVIDSON, J. W. Quick piping: a fast, high-level model for describing processor pipelines. In *LCTES/SCOPES '02: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems* (2002), pp. 175–184.

[27] MISHRA, P., DUTT, N., AND NICOLAU, A. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of the International Symposium on System Synthesis* (October 2001), pp. 256–261.

[28] OPENIMPACT. Web site: http://gelato.uiuc.edu.

[29] PARENT, J. Detecting instruction scheduling constraints, May 2003. Senior Thesis, Department of Computer Science, Hamilton College.

[30] PROEBSTING, T. A., AND FRASER, C. W. Detecting pipeline structural hazards quickly. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (January 1994), pp. 280–286.

[31] RAJAGOPALAN, S., VACHHARAJANI, M., AND MALIK, S. Handling irregular ILP within conventional VLIW schedulers using artificial resource constraints. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (November 2000), pp. 157–164.

[32] RAMSEY, N., AND DAVIDSON, J. Machine descriptions to build tools for embedded systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, Lecture Notes in Computer Science* (June 1998), vol. 1474, pp. 172–188.

[33] SUGA, A., AND MATSUNAMI, K. Introducing the FR500 embedded microprocessor. *IEEE Micro 20* (July 2000), 21–27.

[34] WAHLEN, O., HOHENAUER, M., LEUPERS, R., AND MEYR, H. Instruction scheduler generation for retargetable compilation. *Design & Test of Computers, IEEE 20*, 1 (January 2003), 34–41.

[35] YOTOV, K., PINGALI, K., AND STODGHILL, P. X-ray: A tool for automatic measurement of hardware parameters. In *Proceedings of the 2nd International Conference on Quantitative Evaluation of SysTems* (2005).

[36] ZIMMERMAN, G. The MIMOLA design system: A computer aided processor design method. In *Proceedings of the 16th Annual Design Automation Conference* (1979), pp. 53–58.

[37] ŽIVOJNOVIĆ, V., PEES, S., AND MEYR, H. LISA - machine description language and generic machine model for HW/SW co-design. In *Proceedings of the IEEE Workshop on VLSI Signal Processing* (San Francisco, CA, October 1996).