Instruction Scheduling for Instruction Level Parallel Processors

PAOLO FARABOSCHI, MEMBER, IEEE, JOSEPH A. FISHER, SENIOR MEMBER, IEEE, AND CLIFF YOUNG, MEMBER, IEEE

Invited Paper

Nearly all personal computer and workstation processors, and virtually all high-performance embedded processor cores, now embody instruction level parallel (ILP) processing in the form of superscalar or very long instruction word (VLIW) architectures. ILP processors put much more of a burden on compilers; without "heroic" compiling techniques, most such processors fall far short of their performance goals. Those techniques are largely found in the high-level optimization phase and in the code generation phase; they are also collectively called instruction scheduling. This paper reviews the state of the art in code generation for ILP parallel processors.

Modern ILP code generation methods move code across basic block boundaries. These methods grew out of techniques for generating horizontal microcode, so we introduce the problem by describing its history. Most modern approaches can be categorized by the shape of the scheduling "region." Some of these regions are loops, and for those techniques known broadly as "Software Pipelining" are used. Software Pipelining techniques are only considered here when there are issues relevant to the region-based techniques presented.

The selection of a type of region to use in this process is one of the most controversial questions in code generation; the paper surveys the best known alternatives. The paper then considers two questions: First, given a type of region, how does one pick specific regions of that type in the intermediate code. In conjunction with region selection, we consider region enlargement techniques such as unrolling and branch target expansion. The second question, how does one construct a schedule once regions have been selected, occupies the next section of the paper. Finally, schedule construction using recent, innovative resource modeling based on finite-state automata is then reexamined. The paper includes an extensive bibliography.

Keywords**—***Compilers, instruction level parallelism, instruction scheduling, VLIW.*

Manuscript received February 1, 2001; revised July 24, 2001.

P. Faraboschi and J. A. Fisher are with Hewlett-Packard Laboratories, Cambridge, MA 02142 USA (e-mail: paolo_faraboschi@hp.com; josh_fisher@hp.com).

C. Young is with Bell Laboratories, Murray Hill, NJ 07974 USA (e-mail: cyoung@plang.bell-labs.com).

Publisher Item Identifier S 0018-9219(01)09686-4.

I. INTRODUCTION

This paper is about compiling for instruction level parallel (ILP) architectures, particularly very long instruction word (VLIW) and superscalar architectures. More narrowly, it concerns the code generation phase of compiler back ends, applied to acyclic sections of code. We emphasize the code generation phase because, when writing a compiler for an ILP architecture, code generation differs most from an ordinary compiler. In particular, we focus on code generation for acyclic sections of code.

The rest of this paper is organized as follows.

- Section II discusses the history of the problem, which appeared in disparate domains before it was recognized as a central problem. This history introduces several concepts that are covered in much greater depth in the following sections.

The compiling techniques we discuss are referred to as *region scheduling techniques*, where "region" refers to subsections of a program that are compiled at the same time. ILP compilation techniques differ primarily in the scope of the region considered. This drives the organization of our paper.

- Section III enumerates and describes the major region types that have been suggested.
- Section IV discuss how to form actual regions from a source program, once one has fixed the decision to use a given region type.
- Section V considers the construction and emission of the "schedule," that is, the object program that results from the compilation process. It also discusses a key efficiency issue: determining the available CPU resources while scheduling.

II. HISTORY OF CODE GENERATION FOR ILP PROCESSORS

A. The Three Domains That Drove Research in This Field

During the two decades from 1965 to 1985, three similar problems emerged in three different domains. In each case, an innovative CPU design allowed operations whose execution could overlap in time, within a single instruction stream. Hand coders could often rearrange code to use the CPU efficiently—that is, to make the program overlap more operations and thus complete as quickly as the resources of the machine allowed. However, attempts to automate these code rearrangements (using a compiler or some other code-processing tool) fell well short of what a human could do. The early automated techniques missed many opportunities to overlap operations: at some point in the program execution, there would be operations whose inputs were "ready," there would be sufficient resources to execute the operations, but due to the order in which the code was presented to the CPU, the clock ticked and the operations did not issue. Such failures occurred much less frequently in code reordered by hand, and this gap instigated a research topic.

The three different domains with similar problems were as follows.

1) The compaction of microcode, or the conversion of "vertical microcode" into "horizontal microcode"[1] [1]. This was made more widely applicable by the prospect of "writable control stores" for microcoded CPUs, which were expected to lead to the production of far more horizontal microcode. Extensive printed research on this topic first appeared in the mid-1970s.
2) Compiling for attached signal processing CPUs, or "array processors" [2]. The most famous of these were the Floating Point Systems AP-120b (introduced in 1975) and FPS-164 (introduced in 1980), though there were many others.
3) Code production for the nascent supercomputers' scalar units. These issues first appeared with the delivery of the CDC-6600 in 1963 [3]. However, the problem was treated anecdotally and with *ad-hoc* techniques, rather than as a research topic in its own right.

Each of these types of CPUs can be regarded as the formative stage of a popular type of modern microprocessor. The first led to VLIW architectures, the second prefigured today's DSP cores [4], and the third led to superscalars. Researchers and engineers in these three domains often worked on this problem without knowing of the existence of the other domains.

Today, rearranging code in these three domains is seen as one problem. Historically this was less clear. Unlike VLIWs and DSPs, superscalar hardware sometimes rearranges code at runtime. This led early superscalar architects to maintain that compilers need not schedule code, since the hardware would do the job instead. Few researchers believe this today, rather most feel that the bulk of the rearrangement to be done must be done in advance in each case. Today, superscalar code rearrangements are viewed as ways to compensate for dynamic changes (in latency, for example), or as mechanisms to permit good performance on legacy code when the superscalar microarchitecture changes. Far-reaching adjustments in code order must be made before the code runs.

In most of this paper, we adopt the perspective of filling VLIW-style *instructions*, which contain issue slots for individual *operations*. This clarifies descriptions, since VLIW instruction streams typically map directly to the record of execution, allowing us to avoid considering the variety of rearrangements that superscalar hardware might add. In particular, both VLIW and superscalar designs have issue "slots" that must be filled, regardless of the rearrangements performed by hardware. Filling these slots requires essentially the same considerations for both kinds of architectures.

B. The Problem and the Opportunity: "Filling the Slots"

In each of the environments above, there was the same problem—how to move operations past each other in order to fill slots. That operations must be rearranged is obvious: when an operation cannot be issued in a given cycle, there might be operations below it in the instruction stream that could be. In the case of a VLIW, this means that it might be useful for the compiler to reorder operations, so that an issue slot is not wasted. A glance at any program will convince one that stopping at a "blocked operation" will throw away much of the processor's potential.

At first, the problem of filling slots was seen as relatively straightforward, with natural techniques to address the problem. Although the problem is NP-complete (it is the infamous "job shop scheduling" problem [5]), simple heuristics work in practice. Techniques such as *list scheduling* (defined later) were shown to be effective [6]. We cover these techniques in Section V, but the idea of these techniques is very straightforward. First, build a data precedence graph (or DAG), which represents which operations depend directly on other operations. Then try to account for how critical an operation is when forming the schedule. Operations are considered critical if they have many descendants in the DAG, and/or if they start a long chain of operations which must be executed in turn. When operations are critical, try to schedule them sooner than less-critical operations.

This class of techniques is referred to as "Local Compaction." The earliest known work on the subject appeared in 1971 [7], but the first papers that recognized this problem as a field of study, and put some perspective on it, appeared in 1974–1975 [8]. Research done at the time did not involve realistic implementation in actual products, and some harder problems were largely ignored. For example, there was little consideration of phase ordering with register allocation—though the problem was discussed by Landskov *et al.* [1]. Other largely ignored realities included operations with multiple-cycle latency and complex underlying hardware structures. Landskov *et al.* [1] is also a valuable general survey of local compaction techniques.

1) Local Compaction Did Not Suffice: Local compaction techniques were applied to a very limited region of code: a

[1]Microcode can be thought of as a RISC-like level of code, used to hard-wire a program into the CPU that emulates a more complex instruction set. As language levels, the main difference between vertical microcode and RISC is that microcode is typically more idiosyncratic than RISC. Vertical microcode relates to the hardware at a somewhat lower level than RISC operations, and usually only one program will be written in it, so the price of obscurity is not so high. The horizontal microcode of the 1970s resembles today's VLIW level.

"basic block," namely straight-line code with no branches out except at the bottom, and no branches in except at the top. Intuitively, it did not make sense to schedule across branches, because one would not want to move an operation before a branch when there was a chance that control might transfer to the wrong leg of the branch.

But as these techniques were proposed and investigated, researchers noticed that too many "slots" remained—too many in the sense that hand rearrangements still seemed to use more of what the hardware was capable of. This shortfall occurred because operations from other basic blocks could fill the slots in ways that considered the branch problem and accounted for it. For example, one might shorten a schedule by moving an operation down past a branch, making a copy of the operation in both basic blocks that were targets of the branch. Sometimes this could eliminate a cycle in the earlier block, while not adding cycles to one or both of the target blocks. Many similar motions of this sort become evident after even a few minutes of working with a section of code.

While these opportunities might seem small, experiments [9] had indicated that most of the opportunity for shortening schedules lay beyond branch boundaries. Although few people in the field were aware of these experiments (they were done in an architecture context, not a software tool context), there was the broad realization in the research community that the boundary caused by branches were the largest limiting factor.

This realization motivated many new techniques. The first group of "global compaction" (beyond basic block) techniques could be described as "schedule-and-improve" techniques. These first techniques grew mostly out of the microcode domain, where producing microcode in horizontal format was very difficult, error-prone and time consuming. Schedule-and-improve techniques appeared in the mid-1970s.

The schedule-and-improve techniques worked roughly as follows.

1) The program is divided into basic blocks, each of which is scheduled (local compaction).
2) The scheduled program is then iteratively improved by moving individual operations from block to block.

These techniques gave researchers the intuitive feeling that they were mirroring what people were doing when they did microcode compaction by hand. The best known of the early global compaction techniques was called *MORIF* [10]. MORIF built templates for each operation, where the two-dimensionality of the template allowed a limited representation of the resources used by the operation. After local compaction, a search was done to find candidate moves between blocks, based on the criticality of the operations. Then a catalog of potential legal moves, similar to those shown in Fig. 7, was considered for the most critical operations. There was an additional facility for backtracking to avoid deadlocks that could occur when code motions left an operation no legal slot to occupy.

One fascinating schedule-and-improve technique took the fill-the-slots philosophy to the extreme. Nobel Laureate Ken Wilson [11] and his students, working with FPS CPUs in the attached processor domain, considered a scheme involving Monte Carlo techniques, which had been quite successful in particle physics. The basic idea was that after local compaction, huge quantities of random legal code motions would be tried in search of improvements. Sometimes even bad code motions would be tried, so that hills could be climbed. At the time, many felt that compute time was a basic flaw in this idea. The only report we know of on this work acknowledged "It has not been determined yet how much computer time will be required to achieve effective code optimization by the Metropolis Monte Carlo procedures. If the Monte Carlo approach works . . ."

Schedule-and-improve methods never became popular; nearly all ILP code generation techniques proposed and implemented during the past 20 years have instead involved some form of Region Scheduling.

2) Using Region Scheduling Techniques to Produce Better Schedules: A fundamental flaw in the schedule-and-improve techniques, as seen by some researchers at the time, was that too many arbitrary decisions were made when basic blocks were scheduled. Arbitrary local choices might be all wrong when operations from other blocks were considered. The easiest-to-see example of this is that operations within a block will be scheduled near the beginning of the schedule if possible, but far more critical sequences from later blocks may be the real processing bottleneck. Not starting the later sequences earlier may be an undesirable choice, but by the improve phase, there are no longer slots left in which to start the more critical sequences. Attempting to undo the earlier schedules to make new slots near the top of the blocks can result in impractical computational complexity. Nor does it make sense to reserve slots at the top of every block.

The alternative to schedule-and-improve that has dominated global scheduling since it was introduced in 1979 is called Region Scheduling. The basic idea, first elaborated in the region scheduling technique *Trace Scheduling* [6], is to select code originating in a large region, typically many basic blocks, before scheduling. Then schedule operations from the region as if it were one big basic block.

More carefully, a region scheduling compiler typically follows this sequence in its "scheduler."

1) Given the representation of the code being compiled in the compiler's intermediate form, pick a region from the as-yet-unscheduled code. The region is simply a set of operations originating in more than one basic block. Typically, but not necessarily, a region is a set of contiguous basic blocks. Sometimes a code transformation is done prior to region selection, with the goal of enhancing the region selected, for example by making it larger. Fig. 1 shows an example of a typical code region.
2) Next, place the selected operations on a data-precedence graph. Sometimes the edges or the operations are decorated with additional information that is only important to region scheduling. In addition, some special edges may be added to the graph that prohibit illegal or undesirable code motions. These edges prevent code motions that would be illegal because they

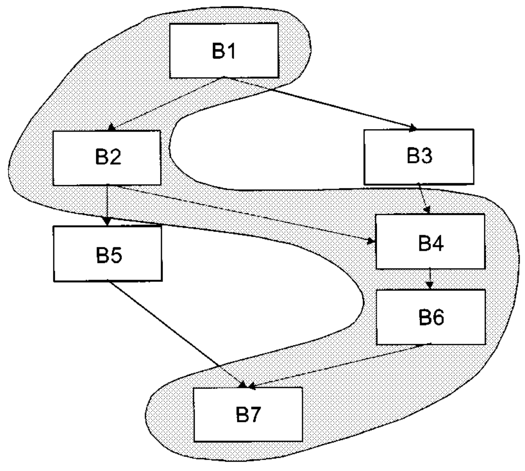

Fig. 1. Example of a typical "region" (in this case, a multiple-entry, multiple-exit region).

would imply a violation in flow control that cannot be compensated for when the schedule is produced. Such special edges are not required for local compaction, since there is no flow control to consider within a basic block.

3) Next, schedule the operations.
4) Finally, either along the way, or in a post-phase, do any necessary fixups. Usually these fixups take the form of extra operations that mitigate what would have been an illegal transformations caused by the positions of the scheduled operations. For example, sometimes an operation is moved in such a way that it should have been executed along some path through the code, but is not. In that case, a copy of the operation is placed in the path from which it is missing. These added operations are often called *compensation code*.
5) Go back to step 1) until no unscheduled code remains.

This is the core algorithm of region scheduling, as specified in Trace Scheduling. There are many other region scheduling algorithms that have appeared in the past 20 years which vary from the Trace Scheduling algorithm at any step, but most follow essentially this framework. Typically, the algorithms vary in the region they select [step 1)] and in the complexity of the schedule construction pass [step 3)].

Section III describes the regions used by the most popular region scheduling algorithms.

C. Software Pipelining

Although region scheduling usually involves regions of loop-free code, most algorithms handle loops in some way. For example, many compilers unroll important loops to increase region size, but, in that case, the region that is actually scheduled is still loop-free.

Software Pipelining is a set of techniques [12]–[14] that deal systematically with scheduling loops. The oldest Software Pipelining techniques we know were developed for the CDC-6600 Fortran compiler, at least as early as 1970 (and probably earlier). We would not be surprised to hear that they were also applied by the IBM compilers for the high-performance CPUs of the same era. Under software pipelining, operations from several loop iterations are gathered in a single new loop. The new loop intermingles operations from different iterations to fill slots. At the entrances and exits of the new loop, new code, analogous to the compensation code of region scheduling, is placed which allows original loop iterations that are not completed in a single iteration of the new loop to be completed.

Interestingly, there has been an idea at the intersection of region scheduling and software pipelining. First suggested by Fisher *et al.* [15], and developed fully by Aiken and Nicolau [16], conceptually this technique unrolls loops an indefinite number of times, and then schedules (using some region scheduling technique) until a pattern becomes apparent. Then a software pipeline is set up, using that pattern as the model of the new loop. This technique has been called "Perfect Pipelining" [17].

D. Predication

ILP is much harder to achieve in the presence of complex control flow. A variety of hardware and software techniques, which fall under the term "predication," transform control dependence into data dependence. We mention predication from time to time throughout the paper, because its presence is required to discuss topics of importance here. Other papers in this Special Issue cover this topic.

E. Terminology

Because of the many origins of instruction scheduling techniques, multiple terms exist to define the same concept. Examples are legion: instructions, operations, syllables, groups, and bundles all refer to the units and groups of scheduling, issuing, and executing code. We will use a set of preferred terms consistently throughout this paper, and we will endeavor to point out synonyms wherever possible.[2]

III. Region Types and Shapes

Most of the alternative methods of region scheduling differ in the shape of the regions they form. Indeed, the algorithms are usually named after the region shape. This section introduces the most commonly implemented regions.

A. Basic Blocks

Basic blocks are a "degenerate" form of regions. Region scheduling algorithms have to work even when their region selection procedures specify a basic block (for example, when there is an unscheduled basic block with no unscheduled contiguous blocks). We mention them here because, despite the generally-held belief that region scheduling is necessary for good performance, its engineering is seen as so daunting that many compilers implement only basic block (local) scheduling, leaving a region scheduling implementation to the indefinite future.

[2]People commonly refer to the phenomenon of inventing new terms to describe existing concepts as one of the manifestations of the NIH ("*Not Invented Here*") syndrome, from which the authors are certainly not immune.

B. Traces

A trace is a linear path through the code. Trace scheduling and its variants are probably the most commonly implemented forms of region scheduling. A Trace consists of the operations from a list $\mathrm{B}_0, \mathrm{B}_1, \ldots, \mathrm{B}_\mathrm{n}$ of basic blocks with the following properties.

1) Each basic block is a predecessor of the next on the list (i.e., for each $k = 0, \ldots, n-1$, B_k falls through or branches to $\mathrm{B}_{\mathrm{k+1}}$).
2) For any i and k, there is no path $\mathrm{B}_\mathrm{i} \rightarrow \mathrm{B}_\mathrm{k} \rightarrow \mathrm{B}_\mathrm{i}$ except for those that go through B_0 (i.e., the code is cycle free, except that the entire region can be part of some encompassing loop).

The outlined area of Fig. 1 shows a typical trace. Note, as the figure shows, this definition does not prohibit forward branches within the region, or flow that leaves the region and comes back into the region at a later point. Indeed, the lack of these restrictions has been controversial in the research community because it makes Trace Scheduling compilers considerably more complex than many researchers feel necessary. Adding those restrictions, sometimes with code duplication to mitigate their impact, is the principle behind several of the later region scheduling regions discussed below.

Traces, and Trace Scheduling, were introduced by Fisher [6], and were described more carefully by Fisher [18], and especially Ellis [19].

C. Superblocks

Superblocks [20] are Traces, with the added restriction that there may not be any branches into the region except to the first block. Thus, a Superblock consists of the operations from a list $\mathrm{B}_0, \mathrm{B}_1, \ldots, \mathrm{B}_\mathrm{n}$ of basic blocks with the same properties as a trace.

1) Each basic block is a predecessor of the next on the list (i.e., for each $k = 0, \ldots, n-1$, B_k falls through or branches to $\mathrm{B}_{\mathrm{k+1}}$).
2) For any i and k, there is no path $\mathrm{B}_\mathrm{i} \rightarrow \mathrm{B}_\mathrm{k} \rightarrow \mathrm{B}_\mathrm{i}$ except for those that go through B_0 (i.e., the code is cycle free, except that the entire region can be part of some encompassing loop).

   And the additional property:

3) There may be no branches into a block in the region, except to B_0. These outlawed branches are referred to in the Superblock literature as *side entrances*.

The restriction against side entrances eliminates the very difficult engineering surrounding compensation code in Trace Scheduling. Superblock formation involves a region enlarging technique called *tail duplication*, which allows Superblocks to avoid ending the moment a side entrance is encountered. Superblocks are built by first selecting a Trace. Then, when a side entrance is encountered, a copy is made of the rest of the trace. The side entrance then branches to this copy, and the new code finally branches to the end of the region, thus eliminating the side entrance. This process continues as more blocks are added to the region and more side entrances are encountered. Tail duplication, in essence, adds the compensation code mandated by an entrance to the region *before* the region is scheduled. As such, it trades space for compiler complexity and must be used selectively. The effect of this on the quality of the resultant schedules is not well characterized in the research. It is worth noting that the originators of Superblock scheduling also proposed region-enlarging techniques that are meant to minimize the extra code involved. These same techniques could be profitably applied to Trace formation and other region selection and enlarging methods.

All of the region scheduling techniques rely on region enlarging techniques to increase the amount of ILP the compiler can exploit. But to a much greater extent, Superblock scheduling relies on tail duplication as an essential feature.

D. Treegions

A Treegion [21] is a region containing a tree of basic blocks within the control flow of the program. That is, a Treegion consists of the operations from a list $\mathrm{B}_0, \mathrm{B}_1, \ldots, \mathrm{B}_\mathrm{n}$ of basic blocks with the following properties.

1) Each basic block B_j, except for B_0, has exactly one predecessor. That predecessor, B_i is on the list, where $\mathrm{i} < \mathrm{j}$. This implies that any path through the Treegion will yield a Superblock, that is, a Trace with no side entrances.
2) For any i and k, there is no path $\mathrm{B}_\mathrm{j} \rightarrow \mathrm{B}_\mathrm{k} \rightarrow \mathrm{B}_\mathrm{j}$ except for those that go through B_0 (i.e., the code is cycle free, except that the entire region can be part of some encompassing loop).

As with superblocks, tail duplication and other enlarging techniques are used to remove side entrance restrictions. Regions in which there is only a single flow of control within the region are sometimes called "linear regions." In that sense, Traces and Superblocks are linear regions, while Treegions are "nonlinear regions."

E. Other Region Shapes

There are several other regions that have been suggested, and some that have been implemented. We do not cover them in detail for various reasons: some because they have only been suggested and leave many of the required implementation details to the reader; others (in particular, Hyperblocks) because they are covered elsewhere, and others because they are scheduling frameworks which have similar goals, but are difficult to describe as scheduling algorithms.

One such method is Trace-2 [22]. Trace-2 is a nonlinear region with a single entrance, like Treegions, but without the restriction on side entrances. The implementation was so difficult that the author gave up in disgust, and knows of no one who has implemented it. The description of Trace-2 leaves many details to the implementer. It is worth noting that the author concluded that a good implementation would require a very thorough use of Program Dependence Graphs [23].

Hyperblocks [53] are single-entry, multiple-exit regions with internal control flow. These are variants of Superblocks with the technique of predication (which requires very extensive hardware support) used to fold multiple control paths into a single Superblock.

Percolation Scheduling [24] is an algorithm for which many rules of code motion are applied to regions that resemble Traces.

Before proceeding, we note that this paper concentrates on region selection in acyclic schedulers. In a cyclic scheduler, region shape is often quite limited, either to a single, innermost loop or to an inner loop that has very simple control flow. These structural requirements mean that a cyclic scheduler can be applied in only a few places, albeit possibly the hot loops of many programs. Profiles will of course show when the cyclic scheduler will actually be beneficial (and conversely, cyclically scheduling a loop with a low trip count can be disastrous). See the other papers in this special issue on cyclic scheduling and the IA-64 architecture for more details about cyclic scheduling.

IV. Region Formation

The previous section introduced a number of region shapes used in instruction scheduling. Once one has decided on a region shape, two questions present themselves: how does one divide a program into regions of a particular shape, and having chosen those regions, how does one build schedules for them? We call the former problem *region formation* and the latter problem *schedule construction*; they are the topics of this section and the next section, respectively. In a sense, the division of instruction scheduling into these two areas indicates the difficulty of the problem or the weakness of the known solutions. One would like to "just schedule" an entire program, but the technology and algorithms do not allow such a direct approach. Instead, schedule construction solves the scheduling problem for those limited cases that we do understand, where the region has a particular shape. Region formation then must divide the general control flow of the program into manageable, well-defined pieces for the schedule constructor to consume.

The combined effects of region formation and schedule construction are critical to performance. Well-selected regions will cover the CFG of the program in a way that keeps the program executing along the expected paths in the scheduled code. Poorly selected regions penalize performance because the schedule constructor will add instructions from infrequently executed parts of the program to the critical execution path. Perhaps stating the obvious, the goal of region formation is to select regions that will allow the schedule constructor to produce schedules that will run well. For this reason, it is important to keep in mind what the schedule constructor will do. The schedule constructor examines only one region at a time, so the goal of region construction is to find frequently executed basic blocks *that execute together* and group them into the same region. If two chunks of the program that execute together are placed in separate regions, then very little benefit will be extracted by instruction scheduling. Designers of region formation passes face three main questions: Which program chunks are frequently executed? How can we tell that two chunks execute together? How does region shape interact with the first two questions?

The traditional answer to the first two questions is to use profiles to measure or heuristics to estimate how frequently each part of the program is executed. Both heuristic and profile-based approaches assign execution frequencies to parts of the program such as nodes or edges in the CFG. In using heuristics and profiles, care must be taken both in the methodology with which the statistics are collected, and in managing the statistics as the program is modified by different parts of the compiler. There has been a variety of innovations in the kinds of profiles collected and in the efficiency of techniques used to collect them in the past decade.

Once one has a usable set of statistics, the question remains how to use them to form useful regions. Region formation often means more than just *selecting* good regions from the existing CFG; it also includes *duplicating* portions of the CFG to improve the quality of the region. Duplication increases the size of the final program, so many different algorithms and heuristics have been applied that make a variety of tradeoffs. Region formation must also produce valid regions that the schedule constructor can use; this may entail additional bookkeeping or program transformations.

The set of options for region formation can be applied in a variety of orders; these phase orders produce an additional set of engineering constraints and tradeoffs.

This section treats the issues roughly in what might be termed, "compiler-engineering order." First, we give an overview of heuristic and profile-based techniques for estimating execution frequencies. Once one has statistics in which one believes, one can try to form regions; the second subsection addresses this topic. We elaborate on enlargement and duplication techniques in the third subsection, then we close with a discussion of phase-ordering issues that relate to region formation.

A. Statistics for Scheduling

All region formation techniques depend on weights assigned to each part of the program. These weights indicate the relative execution frequency of that part. To simplify discussion, we will concentrate on region formation algorithms that work within a single procedure, and therefore we need weights that apply to parts of the procedure's CFG.[3] This section starts by describing the kinds of profile data used for scheduling, continues by summarizing the methods for collecting the profiles, goes on to describe heuristics techniques that avoid profiling, then concludes with a discussion of the bookkeeping subtleties in using profiles.

1) Kinds of Profiles: Prior to 1994, all scheduling-related profiling work concentrated on *points* in the CFG: either nodes or edges in the graph. A node or edge profile would then tell how many times a particular basic block was executed, or how many times control flowed from one basic

[3]One definition of "region scheduling" allows scheduling regions to span procedure calls. For example, see Hank *et al*. [25]. In the interest of brevity, we will not focus on such techniques in this section; the issues remain the same in an interprocedural context.

block to one of its immediate neighbors, respectively. In the past few years, researchers have also started collecting *path profiles*, measuring the number of times a path, or sequence of contiguous blocks in the CFG, was executed. Path profiles have been built in a variety of forms: forward paths, “general” but bounded-length, and whole-program. Each kind of profile has a graph-theoretic definition; comparing the merits and costs of each kind has occupied many researchers. The major differences have to do with trading off the resolution or context associated with each data point with the efficiency of collecting the profile. Each level of additional information allows distinguishing different aspects of program behavior. Advocates of each level have produced optimizations that benefit from this information (e.g., various path optimizations).

One terminological note: the instruction scheduling community refers to the profiles used to drive instruction scheduling as *static branch prediction*. The techniques are identical.

2) Profile Collection: Profiles can be collected in a number of different ways. The oldest technique is *instrumentation*, where extra code is inserted into the program text to count the frequency of a particular event. Instrumentation can be performed either by the compiler, or by a post-compilation tool such as Atom [27]. More recently, hardware manufacturers have added special registers that record statistics on a variety of processor-related events; these registers have been used to perform profiling in tools such as VTUNE [28]. Hardware techniques can have very low overheads, but they do not usually report exhaustive statistics like instrumentation does. Some researchers have built statistically sampling profilers, where an interrupt occasionally examines the machine state. Statistical profiles are noisier than exhaustive profiles, but they also can be collected with extremely low runtime overhead. The Morph project implemented a software-only low-overhead instrumentation system [29], while the DCPI/CCPI project [30] implemented a hybrid system that used both statistical sampling and hardware registers. In an overhead-aware approach that is different from statistical sampling, the Dynamo project from HP [31] implements a lazy version of profiling, where instrumentation is removed after execution exceeds a threshold value.

Because instrumentation usually measures exhaustively, there has been work on efficiently profiling as well. Ball and Larus observed that edge profiling could sample a subset of edges in the CFG but still reconstruct all edge weights [32]. Their technique samples enough edges to break all cycles in the undirected version of the CFG. In later work on forward path profiling, Ball and Larus describe how to enumerate all of the forward paths in a procedure, then determine the path number by modifying a single profiling register as control decisions are made [33]. And the sequence of forward paths can be compressed to describe an entire program trace; Larus calls this a whole-program path [34]. Young describes an efficient collection algorithm for general, bounded-length paths in his thesis [35]; his approach involves lazily exploring the finite-state automaton of all paths in a procedure.

3) Synthetic Profiles (Heuristics in Lieu of Profiles): Historically, there were arguments about whether profiles were legal to use, practical to collect, part of proper benchmarking methodology, and so forth. Heuristic branch prediction assigns weights to each part of the program based solely on the structure of the source program; running the program is not required. The danger of heuristics is that you do not get to see how the program behaves with real data. You may not be able to tell what is common code and what is exceptional code, and your heuristics may then spend valuable optimization time and program space on the uncommon case. But the win is that you might not have to collect statistics on actual running programs, which can seem a daunting operational task.

We know of three major approaches to heuristic profile synthesis: loop-nest depth, weighted heuristics, and neural-network techniques. Loop-nest depth uses standard compiler techniques to find the loops in the program and assign a loop nesting depth. Loop branches are assumed to loop with some fixed probability (typically 90%), and synthetic weights are calculated appropriately [51]. The weighted heuristic technique was pioneered by Ball and Larus [36] and refined by Wu and Larus [37]. Ball and Larus’ original idea was to have a set of heuristics, each of which may or may not apply to a given branch in the program. For example, the loop heuristic predicts that loop branches will stay in the loop, while the return heuristic predicted that returns were not branched to. The heuristics were then empirically ranked for relevance by a training set of programs. Wu and Larus refined this by empirically assigning a branch probability to each heuristic that applied, then using the Dempster–Shaffer formula to blend these probabilities. Lastly, Calder *et al.* trained a neural network based on a corpus of C programs to heuristically predict branches [38]. None of these heuristic techniques gives better results than actual profiling, and no path-based heuristic techniques have been published. But the techniques remain interesting as a way of potentially avoiding building and installing a profiling pass.

Since profiling has been deployed in dynamic optimization, compilation, and translation systems the arguments against profiling have been weakening. Training data sets are now part of standard benchmark suites [39]. Further, in the emerging embedded space, devices are built for a single purpose, and both the common code and sample data sets fall readily to hand. For purposes of discussion, we will refer to both the heuristic and profile-based approaches as producing profiles; one can consider the heuristic techniques to produce synthetic profiles.

4) Profile Bookkeeping and Methodology: If the profiling pass and the instruction scheduling pass are not adjacent in the compiler’s design, then there are bookkeeping issues to manage whenever an intervening pass transforms the program. Further, profiles only measure the parts of the program that were visible before the profiling pass: changes to the CFG made after instrumentation was inserted will not be visible to the profiling code. Furthermore, region formation itself can transform the program in ways that change the applicability of profile information. For these reasons, most textbooks [40], [41] advocate profiling as close to the point where profile information is used as possible. But regardless, bookkeeping issues still must be faced.

Most bookkeeping involves applying what we call *the axiom of profile uniformity*:

> When one copies a chunk of a program, one should equally divide the profile frequency of the original chunk among the copies.

In the case of point profiles, there is probably nothing else to be done. The independent nature of the profile measurements means that no other information remains to disambiguate the copies. However, recent work on path profiling suggests that profile uniformity is a poor assumption to make. Rather, there are correlations among branches in a program [42]–[44], or more succinctly, programs follow paths [45]. Path profiles are not immune to such problems (e.g., one might duplicate an entire path, in which case the profile would not be able to disambiguate between the two copies), but because they capture more dynamic context, they are more resilient to program transformations.

Some researchers have done work on reusing profiles of older versions of a program with the current version of the program; this involves hierarchically matching program components [46]. Another work documents the accounting performed to use a profile from the optimized program code, ideally removing the need to build a separate, profiling version of a program [47].

While not strictly a bookkeeping issue, we note here that cross-validation is crucial to properly studying profile-based optimizations. It is bad methodology to train (profile) and test (evaluate) on the same input, since a real-world application will probably face a variety of inputs. Training and testing on the same input is called *resubstitution* in the learning theory community; it provides a useful upper bound but not a practical performance value. Part of the early debate about using profiles concerned variability across training data sets; Fisher and Freudenberger's study observed that while bad training sets could always be found, the majority of training sets were "reasonable" under cross-validation [48].

Many of the heuristic approaches described above also use a training corpus to derive branch biases or to train the neural network. Such training corpuses are similar to training data sets; results using them without cross-validation should be considered carefully.

B. Region Selection

From this point on, we assume that we have access to execution frequency information. As discussed above, all of the heuristic techniques strive to produce profile-like data without running the program, so the engineering of this and subsequent passes is unaffected by how profiles are collected or synthesized.

The most popular algorithm for region selection is *trace growing* using the *mutual most likely heuristic*. As described before, a trace is a path through the program CFG; it is legal for a trace to have many side entrances and exits. The mutual most likely heuristic works as its name implies. A trace has a first block and a last block; the mutual most likely heuristic can be used to extend either end of the trace. Consider the last block, A, of the trace. Use edge statistics to find its most likely successor block, B. Next, consider B's predecessors. If A is B's most likely predecessor, then A and B are "mutually most likely," and the heuristic adds B to the trace, making it the new end of the trace. The trace can be grown either forward or backward in the CFG. Trace growing stops whenever no mutually most likely block can be found to extend the trace, a back edge is encountered, or the mutually most likely block is already part of a different trace. The process iterates by finding the highest-frequency unselected block in the program and using that as a seed for the trace. The process ends when all blocks have been assigned to traces. Some traces may consist of only their seed block.

The implementers of the Multiflow compiler [51] list a number of alternative heuristics to mutual most likely with which they experimented. None seemed more intuitively satisfying than mutual most likely, and none of them worked better than mutual most likely in practice [52].

Superblock formation involves traditional trace growing, followed by an additional step called *tail duplication*. Tail duplication removes side entrances from all traces by making duplicates of those blocks reachable by side entrances, then reconnecting the side entrances to the duplicates. Once the side entrance edges have been removed from the trace, the trace is left with a single entrance block at its start, making it a valid superblock. Tail duplication increases code size, possibly drastically, so implementers of superblock schedulers take care to schedule only the hot procedures in a program. Tail duplication can also be viewed as an alternative to compensation code that trades code expansion for simplicity in engineering the schedule constructor. See the next section for further discussion of compensation code and suppression thereof.

One unsatisfying aspect of trace formation using point profiles is the cumulative effect of conditional probability. In point profiles, the probability of each branch is measured independently. Whenever the trace crosses a split or join in the CFG, the probability of traversing the entire trace changes. With point profiles, we must assume that this probability is independent for each branch, so the probability of remaining on the trace falls away rapidly. For example, a trace that crosses ten splits, each with a 90% probability of staying on the trace, appears to have only a 35% probability of running from start to end. Researchers have addressed this problem in three ways: building differently shaped regions, using predication hardware to remove branches, and getting better statistics.

Forming nonpath-shaped regions appears to simplify the region selection process, since the region selector can choose both sides of a difficult split or join. However, nonpath-shaped regions require more complicated schedule construction passes or more complicated hardware or both. Complexity is not avoided; it is just handled elsewhere.

Nonlinear region approaches include Percolation Scheduling [24], DAG-based scheduling [49], and Treegions [50].

Predicated execution allows a different approach to difficult (forward) branches: go both ways. Predication necessarily complicates the hardware, the instruction-set architecture (ISA), and the compiler. Hyperblock formation, the most recently documented approach to predication, still uses the mutual-most-likely trace formation mechanism as a basis; it then adds additional blocks to the region based on a heuristic that considers block size and execution frequency [51]. Predication is a very powerful tool for removing unpredictable branches and exploiting otherwise-unused machine resources, but it can also negatively affect performance. The time to execute a predicated block includes the cycles for all scheduled operations, so shorter paths may take longer under predication. Predication is most effective when paths are balanced or when the longest path is the most frequently executed.

Young and Smith explored getting better statistics in a series of papers. Their initial work used global, bounded-length path profiles to improve static branch prediction [54]. Their technique, *static correlated branch prediction* (SCBP), collected statistics about how the path by which a branch was reached affected its direction. By analyzing this information globally, they produced a transformed CFG with extra copies of blocks, but in which the extra copies were statically more predictable. Unfortunately, while SCBP was intended to help back-end optimizations such as scheduling, its transformation of the CFG led to complex graphs in which the branches were individually predictable but in which linear execution traces were hard to find. In later work that directly addressed instruction scheduling, Young and Smith used path profiles to drive the region formation stages of a superblock scheduler [35]. The later work, on path-based instruction scheduling, used a very simple technique to select superblocks. Treat the current trace as a path, and consider its execution frequency. Then consider the execution frequency that results from extending the trace to any of the possible successor blocks. Because the general path profiles include this frequency information, there is no need to invoke profile uniformity. Correlations are preserved through the region formation process.

C. Enlargement Techniques

Region selection alone does not usually expose enough ILP for the schedule constructor to keep a typical wide-issue machine occupied. To further increase ILP, systems use *region enlargement* techniques. These techniques increase the size of the program but also can improve the performance of the scheduled code; using them involves a space-time tradeoff. Many of these techniques exploit the fact that programs iterate; by making extra copies of highly iterated code, more ILP can be found. Such loop-based techniques draw criticism from advocates of other approaches such as cyclic scheduling and loop-level parallel processing, because the benefits of the loop-based enlargement techniques might be found using other techniques. We are aware of no study that has quantified this tradeoff.

The oldest and simplest region enlargement technique is *loop unrolling*. To unroll a loop, make multiple copies of the original loop body, rerouting loop back edges from one copy to the header of the next copy. For the last copy, reroute the loop back edges to the header of the first copy. One is said to "unroll n times" when one makes n extra copies of the loop body. Loop unrolling typically takes place before region selection, so that portions of the larger, unrolled loop body are available to the region selector. In this way, the scheduler can overlap operations belonging to different iterations in the unrolled loop body. Loop unrolling has no awareness of region shape, so it duplicates entire loop bodies without regard to the control flow within the loop. Engineers of trace schedulers do not consider this a problem, as the hot trace through the unrolled code will still be found and the side blocks do not overly burden the schedule constructor. Loop unrolling is often rather effective because a small amount of unrolling is sufficient to fill the resources of the target machine. Loop unrolling is used in most compilers.

The engineers of superblock schedulers take a different approach, forming superblocks before they perform enlarging transformations. They describe three techniques, *superblock loop unrolling, superblock loop peeling*, and *superblock target expansion*. Superblock loop unrolling resembles basic loop unrolling. After superblock formation (but before schedule construction), the most likely exit from some superblocks may jump to the beginning of the same superblock. Such superblocks are called *superblock loops*; unrolling them involves making additional copies of the basic blocks in the superblock and connecting them similarly to the connection in loop unrolling. Superblock loop peeling is similar, but is applied in cases where the profile suggests a small number of iterations for the superblock loop. In such cases, the expected number of iterations are copied, but the last copy is connected to the exit block from the loop, and a special extra copy is made to handle extra iterations not forecast by the profile. Superblock target expansion is similar to the mutual-most-likely heuristic for growing traces downward: if superblock A ends in a likely branch to superblock B, then the contents of superblock B are appended to superblock A to make it bigger.

Young and Smith's path-based approach to superblock selection also lends itself to superblock enlargement. The same algorithm does both formation and enlargement: grow downward only, and choose the most likely successor block. A number of thresholds stopped growing traces: low likelihood in the successor block, low overall likelihood of reaching the end of the trace, and sufficient number of instructions in the trace. General path profiles provide exact execution frequencies for paths within the bound of the profiling history depth. Young and Smith found modest performance improvements from using path profiles in addition to the engineering simplifications already described.

D. Phase Ordering Considerations

Just within the region formation pass(es), there are phase ordering considerations. The designers of the Multiflow compiler chose to place enlargement (loop unrolling) before trace selection. The superblock-based techniques chose and formed superblocks before enlarging them. Neither is clearly preferable, but they are determined by the engineering constraints of the chosen approach.

Other optimizations also have phase ordering interactions with region formation. Other ILP-enhancing optimizations, such as dependence height reduction [55] should be run before region formation. Hyperblock-related techniques have an entire suite of transformations including if-conversion and reverse if-conversion that we do not discuss here.

V. Schedule Construction

The previous section described techniques for selecting and enlarging the individual compilation regions. This section discusses assigning operations in the selected region to units on the target machine and time slots in the schedule. A *schedule* is thus the set of annotations that indicate unit assignment and cycle time of the operations in a region. A *schedule constructor* or *scheduler* is the phase that produces such a schedule. Like region formation, schedule construction techniques vary depending on the shape of the region being scheduled: different kinds of regions require different transformations.

The goal of any scheduling algorithm is to minimize an objective cost function while maintaining the semantics of the program and obeying the resource limitations of the target hardware. In most cases, the objective function is the *estimated completion time* of the region, although it is also possible to find domains that demand more complex objective functions. For example, a scheduler for an embedded target may add *code size* or *energy efficiency* to the objective cost function. This section of the paper concerns itself with schedule construction while maintaining the semantics of the program. The last part of this section, Section V-F, describes two competing approaches to resource management under scheduling.

Depending on the ISA and microarchitecture of the target machine, different schedules may maintain or violate the semantics of the original program. Different machines can have different amounts of hardware support or checking to support ILP. Some of the alternatives include the following.

- *Visible versus hidden latencies:* Basic operations vary widely in complexity; some execute in multiple cycles and produce their results some cycles after their issue time (e.g., divides typically require multiple cycles). When the ISA exposes visible nonunit latencies to the compiler (most VLIW machines fall into this category), an erroneous latency assumption in the scheduler may change the semantics of the program. Alternatively, the microarchitecture can independently check latency assumptions (through some form of scoreboarding technique, commonly used in superscalars) and correct violations through stalls or dynamic rescheduling. In this case, the compiler can assume average or worst-case latencies without additional work to maintain program correctness.
- *Explicitly parallel versus superscalar:* Machines that include instruction-level parallelism may choose to expose it in the ISA (VLIW), or to hide it with sequential instruction semantics and let the hardware rediscover it at runtime (superscalar). In the former case, the compiler must monitor machine resources to avoid generating illegal code. In the latter case, the compiler may estimate resource usage for performance reasons, but need not monitor them for legality. However, the compiler may still need to follow certain encoding rules to ensure that the underlying implementation is able to find the parallelism that the compiler has discovered. For example, all Alpha implementations can issue multiple instructions per cycle, but all of the instructions must come from the same cache line for them to be able to issue in parallel.

Recently introduced EPIC architectures blend the explicitly parallel and superscalar approaches. Such ISAs closely resemble VLIWs, but they also allow limited sequential execution within a parallel execution unit (called an issue group) to accommodate the resource limits of different implementations. This blended approach allows binary compatibility over a set of implementations, but still allows much of the ILP extraction to be done by software.

This section begins by describing how schedulers analyze programs. Armed with the analysis techniques, we then describe a number of approaches to compaction, or actual schedule construction, in Section V-B. Maintaining program semantics (or correctness) is treated next. Then we discuss clustering, a microarchitectural technique of increasing necessity that further complicates code generation. We continue with two broader perspectives, the first on the interactions of phases in code generation, and the second on other optimization (in the true, not compilation-only sense) approaches to scheduling. Lastly, we discuss managing resources during scheduling.

A. Analyzing Programs for Schedule Construction

Dependences (sometimes also called *dependencies*) are sequential constraints that derive from the semantics of the program under compilation. Dependences prohibit some reorderings of the program. Program dependences come in two kinds: *data* and *control* dependences. The data flow of the program imposes *data dependences*, and similarly the control flow of the program imposes *control dependences*.

Data dependences come in three types: *read-after-write* dependences, *write-after-read* dependences, and *write-after-write* dependences. The first kind, read-after-write dependences (also called *RAW*, *flow* or *true*), occurs when one operation uses the result of another; reordering would break this flow of data in the program. Write-after-read dependences (also called *WAR* or *anti* dependences) occur when one operation overwrites a value after it has been used by another operation; reordering would overwrite the correct value before it is used. Third, write-after-write dependences (also called *WAW* or *output* dependences) occur when two operations write to the same location; reordering the operations

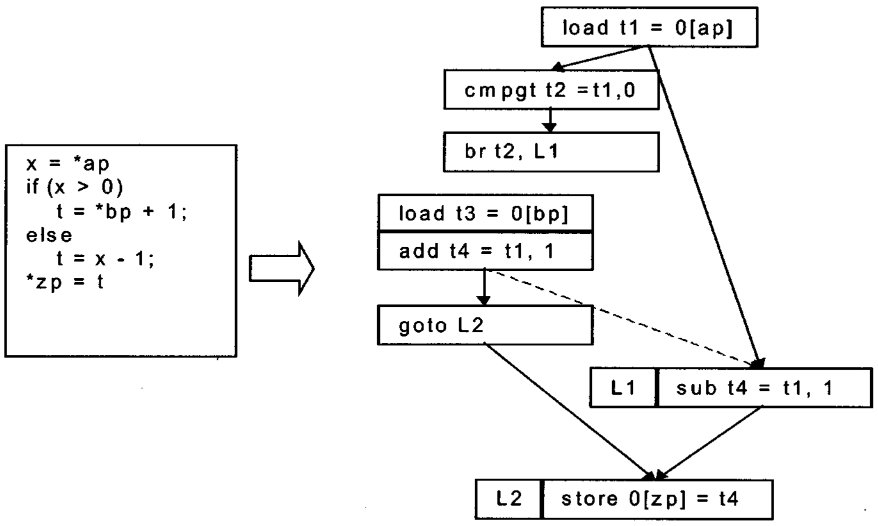

Fig. 2. Example of data dependences. Solid arcs are flow dependences, dashed arcs are output dependences.

will cause the wrong value to be the final value. The latter two kinds of dependences, WAR and WAW, are also called *false* dependences because they can be removed by renaming (adding extra temporary variables to the program). Fig. 2 shows an example of data dependences expressed as arcs between operations.

Control dependences represent constraints imposed by the control flow of the program. For example, if an instruction can only be reached by passing through a particular conditional branch, then that instruction is control-dependent on the branch. For basic blocks, control dependences only affect the scheduling of the single entrance and single exit of the region. For more complex region types (with multiple entrances or exits), control dependences may also constrain scheduling, preventing certain operations from executing before an entrance (*join*) or after an exit (*split*). Once the program reaches the scheduler, the compiler usually represents the control dependences as arcs connecting control-dependent pieces. For example, Fig. 3 shows the control dependences among basic blocks in a simple program.

As described above, both data and control dependences are constraints between pieces of the program that make some reorderings illegal. These constraints induce a partial ordering on the pieces (whether the pieces are instructions or basic blocks), and any partial ordering can be represented as a directed acyclic graph (DAG). Such graphs of dependences are used frequently in scheduling analysis passes; variants (depending on how the graphs are built) are known simply as the DAG, as the *data dependence graph* (DDG), or as the *program dependence graph* (PDG). All such variants are graphs where nodes represent operations and arcs are the data-dependence constraints among them; building the variants typically requires quadratic time complexity in the number of operations.

The DAG represents the constraints that the scheduler must obey to maintain program semantics. But it also

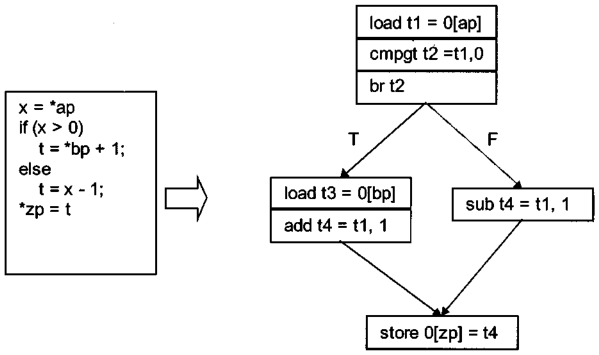

Fig. 3. Example of control dependences among basic-blocks.

includes information that allows the scheduler to evaluate the relative order and importance of the operations. After building a DAG, we can label operations with some interesting properties. Two obvious ones are *depth* (the length of the longest path from any root of the DAG) and *height* (the length of the longest path to any leaf of the DAG). Operations for which *depth* equals *max_height−height* are on the critical path of the region. For noncritical operations, the range [*depth*, (*max_height−height*)] are the time slots in the schedule where the operation can be placed without increasing the schedule length.

The intermediate representation adopted by the upstream phases of the compiler may choose various methods to represent dependences, such as virtual register names, arcs of an SSA (static single assignment) web, memory references, and so on. The case of memory references is particularly interesting: unlike other types, the schedule must often make conservative assumptions for memory references. *Alias analysis* is the set of techniques that help the scheduler *disambiguate* among memory references.

Most scheduling decisions obey the constraints found in the DAG. However, some of the most powerful scheduling

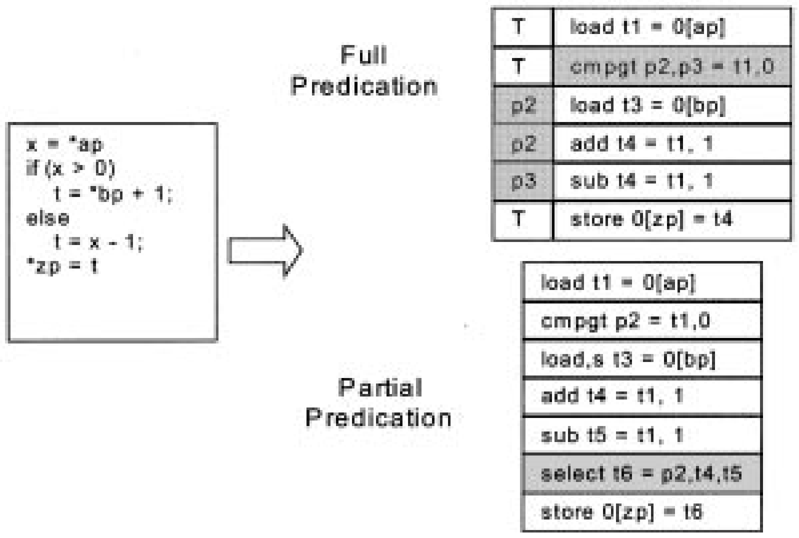

Fig. 4. Example of fully and partially predicated code. In the full-predication case, the *cmpgt* operation produces the true (p2) and false (p3) predicate at the same time. A predicate T means "always executed." In the partial-predication case, we use a *select* operation to implement the "$a = b?c : d$" function. Note that partial predication usually requires speculation to be effective (the *load* operation becomes a speculative *load.s* operation).

techniques aim to relax or remove dependences. Two fundamental techniques are employed by schedulers (or in the preceding phases) to transform or remove control dependences.

- *Predication* (also called *if-conversion* and *conditional execution*), converts multiple regions of a control flow graph into a single region composed of *predicated* (*conditional*) code. In other words, predication transforms control dependences into data dependences. In the case of *full predication*, instructions take additional operands that determine at run-time whether they should be executed or ignored (treated as nops). These additional operands are called *predicate operands* or *guards*. In the case of *partial predication*, special operations (such as *conditional move*, or *select*) achieve similar results. Fig. 4 shows the same code used in Fig. 3, after predication. The case of full predication is particularly relevant to region compaction, since it can affect how the scheduler chooses units and allocates registers. Partial predication, for example using *selects*, is a more natural fit to the scheduling phase and we will not treat it separately.
- *Speculative code motion* (or *code hoisting* and sometimes *code sinking*), moves operations above control-dominating branches. Note that this transformation does not always preserve the original program semantics, and in particular, it may change the exception behavior of the program. A compiler may only introduce speculative operations when certain conditions are satisfied, depending on the degree of ISA support for the execution of speculative memory operations and in general on the exception model imposed by the runtime system. Note that, unlike predication, speculation actually *removes* control dependences, thus potentially reducing the critical path of execution.

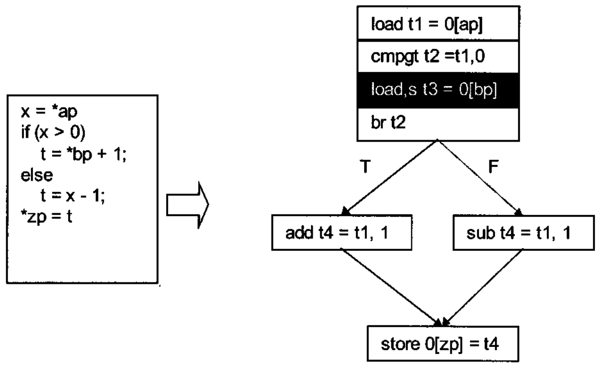

Fig. 5. Example of speculative code motion (compare it with Fig. 3). The load operation becomes speculative (marked load.s), once we move it above a branch.

Fig. 5 shows an example of speculative code motion. As another example, a *trace scheduler* may be able to move operations either above splits or below joins, knowing that a successive *bookkeeping* phase will generate compensation code to re-establish the correct program semantics.

After regions have been formed, classical optimizations (e.g., constant propagation and partial redundancy elimination) can often serve to remove operations and their corresponding data dependences. Also, there is a class of *dependence-height reducing optimizations* [55].

B. Compaction Techniques

This subsection briefly reviews some of the most widely used scheduling techniques for ILP targets. Research and production compilers in the past 20 years adopted many different approaches, making exhaustive description impossible. Instead, we enumerate techniques that are components of almost all approaches to compaction.

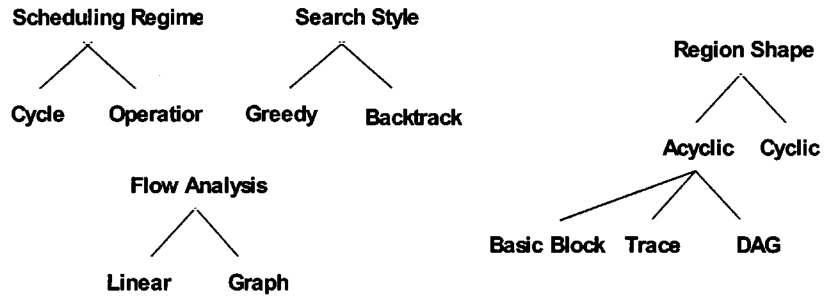

Fig. 6. A set of decision trees characterizing compaction techniques.

We classify compaction techniques according to different features, such as: cycle versus operation scheduling, linear versus graph-based analysis, acyclic versus cyclic regions, and greedy versus backtracking search. Fig. 6 shows a set of decision trees characterizing compaction techniques. This paper largely focuses on *greedy* scheduling techniques for *scalar* (acyclic) regions, but this section touches briefly on the set of alternatives.

1) Cycle Versus Operation Scheduling: As discussed above, the goal of the scheduler is to allocate operations to cycle slots while minimizing an objective function. The two parts of this allocation, operations and time slots, suggest two approaches to overall scheduler strategy: *operation-based* and *cycle-based* scheduling.

- *Operation scheduling* repeatedly selects an operation in the region and allocates it in the earliest cycle that dependences and target resources allow. Operation scheduling techniques vary based on the selection method, which can be guided by a number of heuristics or priority schemes.
- *Cycle scheduling* (sometimes unfortunately called *instruction scheduling*) repeatedly fills a cycle (usually corresponding to an issue group) with operations selected from the region, proceeding to the next cycle only after exhausting the operations available in the current cycle.

Operation scheduling is theoretically more powerful than cycle-based scheduling, but it is much more complicated to engineer, especially for complex regions that extend beyond basic blocks. We are not aware of any production compilers that use operation scheduling.

2) Linear Techniques: The simplest schedulers use linear techniques [1]. When compile time is of utmost importance, linear methods have the major advantage of having $O(n)$ complexity, for a region composed of n operations. Techniques that use data-dependence graphs have at least $O(n^2)$ complexity. However, for practical region sizes and machine speeds, differences are often not significant enough to justify the performance loss of an inferior technique. For this reason, graph-based techniques have almost universally replaced linear techniques in modern compilers.

Most of the linear techniques use either or both.

- *As-Soon-As-Possible* (ASAP) scheduling, where we place operations in the earliest possible cycle that resource and data constraints allow through a single top-down linear scan of the region. Note that a graph is not necessary to enforce data dependences, but a simple time-annotated table of produced values suffices.
- *As-Late-As-Possible* (ALAP) scheduling, where we place operations in the latest possible cycle that resource and data constraints allow through a single bottom-up linear scan of the region.

For example, Critical-Path (CP) scheduling uses an ASAP pass followed by an ALAP pass to identify operations in the critical path of the computation (those that have the same cycle assignment in both schedules). Remaining noncritical operations are allocated in a third linear pass.

3) Graph-Based Techniques (List Scheduling): The major limitation of linear techniques is their inability to make decisions based on global properties of the operations in the considered regions. Such global properties are incorporated in the DAG described above. Most of the scheduling algorithms that operate on DAGs fall into the category of *list scheduling*. List scheduling techniques work by repeatedly assigning a cycle to an operation without backtracking (greedy algorithms), and efficient implementations have $N\log(N)$ computational complexity (in addition to the DAG creation, which has N^2 complexity).

List scheduling repeatedly selects an operation from a *data-ready queue* (DRQ) of operations ready to be scheduled. An operation is *ready* when all of its DAG predecessors have been scheduled. Once scheduled, the operation is removed from the DRQ and its successors that have become ready are inserted. This iterates until all operations in the region are scheduled. The performance of list scheduling is highly dependent on the order used to select the scheduling candidates from the DRQ, and—in case of cycle-based scheduling—on the scheduler's greediness.

We can tackle the first problem (DRQ order) by assigning a priority function to each operation in the DAG. Ideally, the *height* of the operation is what should drive the priority function: at any point during scheduling, operations with the greatest height are the most critical. However, depending on other scheduling considerations, compilers have also used the *depth*, a combination of the two, or a depth-first topological sorting of operation within a connected component of a region. In general, all these choices are based on heuristics, and have to strike a balance among schedule quality, implementation complexity and compile-time performance. Chekuri *et al.* [60] survey priority functions.

The second problem (greed control) is more complex, since cycle-based schedulers often tend to be too greedy. Greediness may hurt performance in two ways.

1) Operations that we schedule too early and that occupy resources for multiple cycles may prevent more critical operations that become ready later to be scheduled due to resource constraints.
2) Operations that we schedule too early may unnecessarily increase register pressure and force spills that could be avoided if we had decided to delay their schedule.

Unfortunately, most workarounds for this problem rely on heuristics that rarely apply outside of the domain where they were introduced.

4) Loop Scheduling: Many programs spend most of their time in loops. Therefore, a good loop scheduling strategy is a fundamental component of an optimizing compiler. The simplest approach to loop scheduling, *loop unrolling*, was mentioned in Section IV-C on region enlargement. Loop unrolling does not actually allow acyclic schedulers to handle loops; rather, it enlarges the acyclic part of a loop to allow the acyclic scheduler room to work. More sophisticated techniques directly address scheduling loops and their back edges.

Software pipelining is the class of global cyclic scheduling algorithms, which exploit inter-iteration ILP while handling the back-edge barrier. Within software pipelining algorithms, *modulo scheduling* is a framework that produces a *kernel* of code that sustainably overlaps multiple iterations of a loop. The kernel is built so that neither data dependence nor resource usage conflicts arise. Correctly entering and exiting the kernel code is handled by special code sequences called *prologues* and *epilogues*, respectively; they prepare the state of the machine to execute the kernel and correctly finish executing the kernel and recording its results. Prologue and epilogue code is analogous to compensation code generated by acyclic schedulers: it is necessary to maintain correctness but causes code expansion. Hardware techniques can reduce or remove the need for prologues and epilogues.

Modulo scheduling efficiently explores the space of possible kernel schedules to find the shortest legal one. The length of the kernel, which is the constant interval between the start of successive kernel iterations, is called the *initiation interval (II)*. The resources required by the operations in a loop and the inter-iteration data dependences in a loop place lower bounds on the II; these are called the *resource-constrained minimum II* (ResMII) and the *recurrence-constrained minimum II* (RecMII), respectively. To enforce II-derived resource constraints, the modulo scheduler uses a *reservation table* for the machine resources that checks for conflicts not just in the current cycle, but also in all schedule cycles that differ by II from the current cycle. The scheduler begins searching schedules at the higher MII, and heuristics guide whether to continue searching, to backtrack, or to abandon the current II for the next higher one. In practice, iterative modulo scheduling generates near-optimal schedules (optimality in about 96% of the observed loops). In addition, its compile-time performance is good, and it is often much more efficient than any other cyclic or acyclic scheduler based on loop unrolling.

On the downside, modulo scheduling is most effective on well-structured single loops. Nested loops can be handled by recursively invoking the modulo scheduler, but outer loops must then include the prologue and epilogue code of the inner loop. Loops with exits can be handled, but at the expense of a much greater complexity. Control flow in the loop body (e.g., a single if–then–else) can be handled only with great difficulty; most approaches rely on some form of predication to if-convert the loop body.

C. Compensation Code

Under the term "compensation code," we cover the set of techniques that are necessary to restore the correct flow of data and control because of a global scheduling phase or a global code motion across basic blocks. Depending on the shape of the region that the compiler adopts, the complexity of the task of generating compensation code varies from trivial to extremely complex.

- Scheduling techniques that primarily deal with basic blocks and move operations around them generate compensation code as part of the code motion itself.
- Superblock techniques generate compensation code as part of the tail duplication process, as we described in the previous sections.
- Trace scheduling (as well as other techniques that allow multiple-entry regions) involves a more complex bookkeeping process, since the compiler is allowed to move operations above join points, as well as move branches (split points) above operations that were below them in the original program sequence.

A complete discussion of all the intricacies of compensation code is well beyond the scope of this paper. However, compilers base many of the compensation techniques on a variation of a few simple concepts that we illustrate in the following. When the compiler schedules a region, and is allowed to move operations freely with respect to entries and exits, we can identify four basic scenarios (Fig. 7).

1) *No Compensation*, Fig. 7(a): This happens when the code motions do not change the relative order of operations with respect to joins and splits. This also covers the case when we move operations above a split point, in which case they become *speculative*, as we discussed in the previous sections. The generation of compensation code for speculative code motions depends on the recovery model for exceptions. In the case of *nonrecovery speculation* (also called *silent speculation* or *dismissible speculation*), no compensation code is necessary. In the case of *recovery speculation*, the compiler has to emit a recovery block to guarantee the timely delivery of exceptions for correctly speculated operations.
2) *Join Compensation*, Fig. 7(b): This happens when an operation B moves above a join point A. In this case,

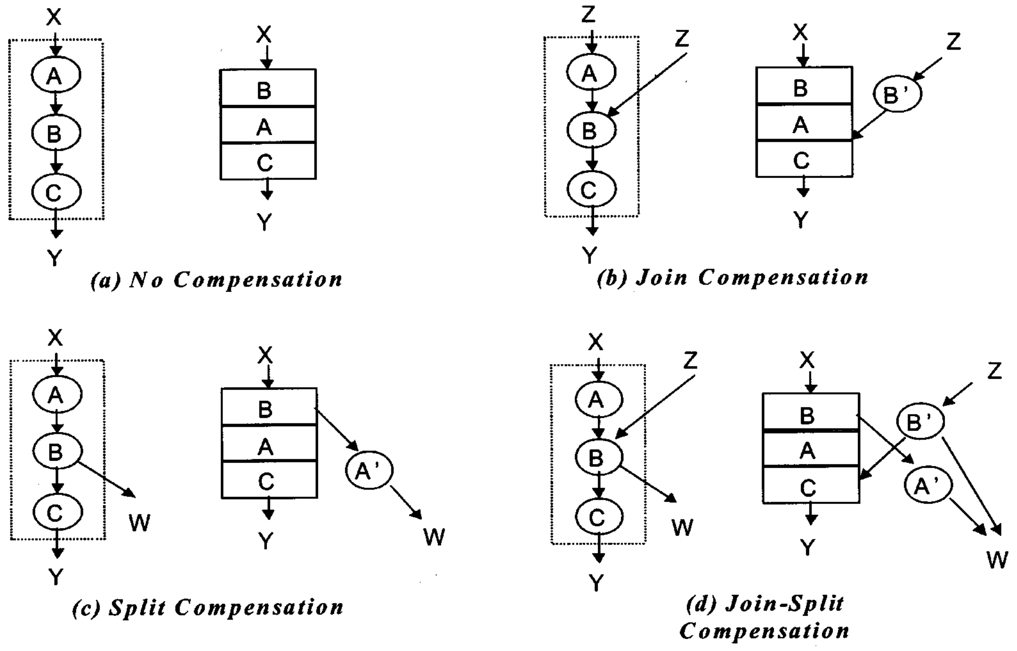

Fig. 7. The four basic scenarios for compensation code. In each panel, the picture on the left represents the original control flow graph, with the selected region to be compacted. The picture on the right represents the compacted schedule (with A moved above B), and the compensation code added to the resulting flow graph to restore correctness.

we need to drop a copy of operation B (called B′) in the join path. A successive phase of the region selector picks operation B′ as part of a new region and schedules it accordingly. Note that, if operation B is only *partially moved* above the join point A (this can happen for multi-cycle operations in explicitly scheduled machines), then we only need to partially copy B to the join path. Also, in this case (called *partial schedule*) the partial copy must be constrained to be scheduled exactly before the join point.

3) *Split Compensation*, Fig. 7(c): This happens when a split operation B (i.e., a branch) moves above a previous operation A. In this case, the compiler produces a copy of A (called A′) in the split path. The same scheduling considerations from the join case apply to the split case: A′ is successively picked as part of another region, unless it is a partial copy, in which case is constrained to happen right after the split.
4) *Join-Split Compensation*, Fig. 7(d): Cases that are more complicated appear when we allow splits to move above joins (presented in the figure), or splits above splits. For example, if we move a split B above a preceding join A, in addition to having to copy A to A′, we also need to create a copy of the branch B′ to target the split destination, to guarantee the correct execution of the $Z \rightarrow B \rightarrow W$ path.

In general, the rule to keep in mind when thinking about compensation code is to make sure that we preserve all paths from the original sequence in the transformed control flow after scheduling. Obviously, the order of operations may be different, but we nonetheless need to execute all the operations from the original control flow. For example, the copy B′ in (b) restores the $Z \rightarrow B' \rightarrow C \rightarrow Y$ path; the copy A′ in (c) restores the $X \rightarrow B \rightarrow A' \rightarrow W$ path, and so on. If certain conditions apply, it is possible to optimize (that is, inhibit) the generations of compensation copies. For example, in (c) we do not need to copy A′ to the split path if A has no side effects and the values produced by A are not live at the exit point W.

D. Clustering

ILP architectures have high register demands. Each parallel execution unit typically consumes two operands and produces a third, requiring a large, multiported register file to support even narrow-issue machines. *Clustering* [56] provides a natural solution to these problems. A clustered architecture divides a multiple-issue machine into separate pieces (obviously called *clusters*), each consisting of a register bank and one or more functional units. Functional units can efficiently access the local registers in their associated bank. Depending on the architecture, remote registers may be directly addressable or they may only be reachable using intercluster move instructions. Regardless of how remote access is specified, it is typically slower than local access and is often subject to resource limitations. Fig. 8(b) shows a 4-issue ILP datapath with an 8-read, 4-write central register. Fig. 8(a) shows the related clustered 4-issue ILP datapath, with two clusters of two function units and one register bank each. Fig. 8(a) also shows the communication link between the clusters.

Clustering complicates compilation. As long as the clusters are architecturally visible, the compiler must place operations to minimize intercluster moves and unbalanced use of the clusters. This is a new compiler responsibility, as

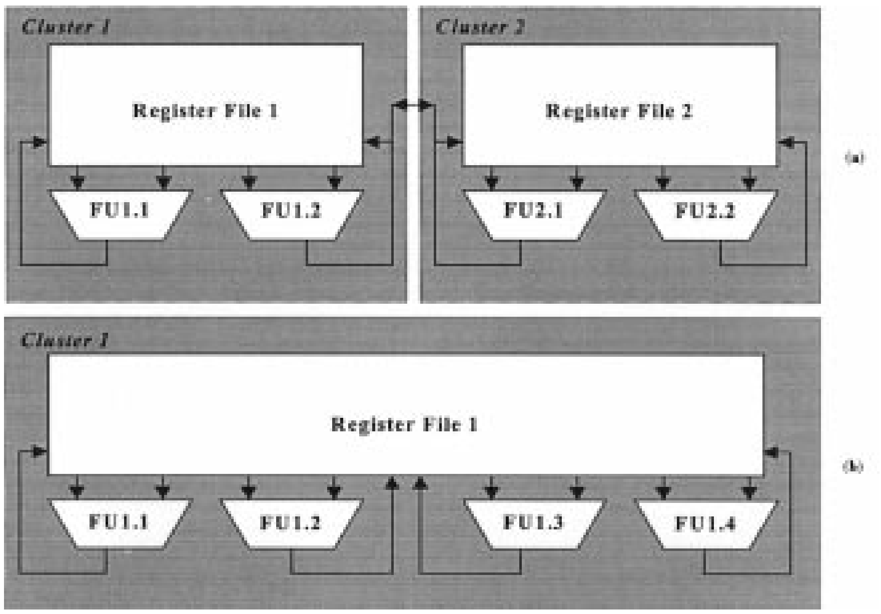

Fig. 8. Clustered VLIW architectures.

the decision about *where* to execute an operation was traditionally either empty (e.g., for a scalar machine) or handled transparently by the hardware (e.g., for a completely connected superscalar machine). The difficulty of the compiler problem depends on the way that the ISA specifies communication among clusters. When the hardware transparently supports fetching remote operands (possibly with a dynamic penalty), the compiler's task is to minimize the number of dynamic stalls. In this case, compiler choices can degrade performance, but correct code will always be generated. When connectivity is architecturally exposed (either by explicit intercluster moves or by limited ways to specify remote registers), the compiler must issue copy operations to move data to appropriate locations. In this case, compiler choices affect correctness on top of performance.

Although some recent approaches (such the *unified assign-and-schedule* technique) advocate a unified clustering and scheduling step, most compilers implement clustering before scheduling. The clustering phase preassigns operations to clusters, then the scheduling phase assigns operations to functional units within clusters. As examples, we outline two preassignment techniques, *Bottom-Up-Greedy (BUG)* and *Partial Component Clustering (PCC)*.

BUG was originally designed for the Bulldog compiler [19] at Yale in the mid-1980s. BUG has two phases.

1) BUG first traverses the DAG from the exit nodes (leaves) to the entry nodes (roots), estimating the likely set of functional units to be assigned to a node based on the location of previously assigned operands and destinations. When it reaches the roots, BUG works its way back to the leaves, selecting the final assignment for the nodes along the way. To reach a final assignment, BUG estimates the cycle in which a functional unit can compute the operation based on resource constraints, the location of the operands and the machine connectivity. Once the cycle estimates for all the feasible units for a node are available, BUG selects the unit producing the smallest output delay for each node.
2) In the second phase, BUG assigns initial and final locations to the variables that are live in and out of the DAG. This phase is quite delicate, since it affects the adjoining regions of code, and particular care must be taken to avoid redundant duplication of locations for critical values (such as induction variables in loops) without sacrificing the parallelism opportunities.

BUG makes a few simplifying assumptions in its operating mode. First, functional units are the only limiting resources in the machine; conflicts over register-bank ports or buses are ignored. Second, resource costs and delays for scheduling explicit copy instructions are ignored. Third, register pressure is ignored. Under register pressure, the topology of the DAG can change significantly due to the presence of spill/restore operations. This is one of the major limitations of the algorithm.

Scheduling a DAG of 1000 operations for a machine with four symmetrical clusters implies 4^{1000} clustering combinations. As an alternative to BUG, consider reducing the dimensionality of the problem and applying some form of componentization. To do this, construct "*macro-nodes*" for partially connected components of the original DAG. These components can then be treated as indivisible units and assigned to a single cluster. PCC adopts this philosophy. It works in three phases.

1) *Partial Component Growth*, where the compiler assembles groups of operations from the DAG into "components" (macro-nodes), based on connectivity criterion.
2) *Initial Assignment*, where we perform a greedy BUG-style pass to produce a reasonable cluster assignment for the components.

3) *Iterative Improvement*, where pairs of cluster assignments for components are swapped repeatedly until we meet a termination criteria.

Various experiments suggest this rule of thumb: breaking the CPU into two clusters costs around 15%–20% lost cycles; breaking into four clusters costs around 25%–30%. Whether these results approach the theoretical limits is open research. Direct computation of optimal clustering remains infeasible.

E. Phase Ordering of Register Allocation and Scheduling

Register allocation and scheduling have conflicting goals. A register allocator tries to minimize spill and restore operations, creating sequential constraints (for register reuse) between operations that the scheduler could otherwise place in parallel. A scheduler tries to fill all the parallel units in the target machine and may extend variable lifetimes by speculative code motion. Both of these changes increase register pressure. Register allocation and scheduling must coexist in the compiler, but how to order them is not apparent *a priori* [57]–[59]. These are some of the alternatives.

- *Instruction Scheduling followed by Register Allocation* values exploiting ILP over register utilization. It assumes enough registers are usually available to match the schedule. This method encounters problems when the scheduler increases register pressure beyond the available registers. This can happen on wide-issue machines in regions with considerable amounts of ILP (e.g., scientific code or multimedia applications). Note that the register allocator must insert spill and restore code in the already scheduled code, which can be difficult task on statically scheduled targets. This technique is common in product compilers for modern RISC processors.
- *Scheduling followed by Register Allocation followed by Post-Scheduling*. This variation of the previous technique adds a *post-scheduling* pass after register allocation. The post-scheduler rearranges the code after spill code has been placed and register assignments are final. To guarantee convergence, the post-scheduling phase cannot increase the number of required registers. This approach makes a good engineering/performance tradeoff and is common in industrial compilers.
- *Register Allocation followed by Instruction Scheduling* prioritizes register use over exploiting ILP. This technique works well for target machines with few available registers (such as x86 architectures). However, the register allocator introduces additional dependences every time it reuses a register. This leads to very inefficient schedules.
- *Combined Register Allocation and Instruction Scheduling* attempts to build a single pass that trades off spill costs against lost ILP. Although potentially very powerful, this approach involves the most engineering complexity. Since register resources may never be freed, a straightforward list-scheduling algorithm may not converge, and additional measures are necessary to ensure that scheduling terminates. The DAG changes dynamically with the addition of spill and restore operations. These changes affect operation heights, operation depths, and the critical path of the region, possibly invalidating previously made choices. Finally, the integrated scheduler-allocator must handle values that pass through but are not used in the region. A mechanism called "delayed binding" (to defer the choice of a location of a value until needed) addresses this problem, but further complicates the scheduler.

Finally, it is worth mentioning *cooperative approaches* where the scheduler monitors the register resources of the target and estimates register pressure in its heuristics. A post-pass register allocator adds spill and restore code when needed. This approach is particularly promising, since the scheduler remains simple but the system can still avoid pathological register pressure cases.

F. Another View of Compaction Problems

Scheduling problems are not unique to compilers. In fact, the entire field of Operational Research (OR) is dedicated to solving scheduling problems. From an OR viewpoint, region compaction is very similar to many *job-shop scheduling* (JSP) problems from the manufacturing realm. In JSP, a finite set of machines process a finite set of jobs. Each job includes a fixed order of operations, each of which occupies a specific machine for a specified duration. Each machine can process at most one job at a time and, once a job initiates on a given machine, it must complete uninterrupted. The objective of the JSP is to find an assignment of operations to time slots on the machines that minimizes the maximum completion time of the jobs.

Deterministic representation and techniques that apply to JSP problems include the following.

- Mixed integer linear programming (MIP). MIP represents the problem as a linear program with a set of linear constraints and a single linear objective function, but with the additional restriction that some of the decision variables are integers. The simplex method is one of the best-known algorithms for MIP problems.
- Branch-and-Bound techniques dynamically explore a tree representing the solution space of all feasible sequences. Bounding techniques prune the search space. Tight bounds are critical to the convergence of the algorithm to a good solution.
- Iterative improvement methods, starting from an initial legal solution, optimize a cost function by exploring neighboring solutions. These are analogous to such techniques used to solve max-flow/min-cut problems in other domains.
- Approximation methods, bottleneck-based heuristics, constraint satisfaction AI techniques, neural networks, adaptive searches, hybrid approaches, iterative improvement, are other techniques that have been proposed.

- Non-deterministic iterative methods include techniques like:
  - Simulated Annealing (SA), a random oriented search technique introduced as an analogy from the physics of the annealing process of a hot metal until it reaches its minimum energy state.
  - Genetic Algorithms (GA), based on an abstract model of natural evolution, where the quality of individuals improves to the highest level compatible with the environment (constraints of the problem).
  - Tabu-Search (TS), based on intelligent problem solving.

Why have compiler writers not extensively used these techniques? In general, OR techniques seek optimal or near-optimal solutions and are designed to solve small numbers of large-scale problems. On the other hand, a compiler can often compromise optimality for the sake of compile speed, and it needs to solve a very large number of comparatively small problems. We omit more details of OR techniques; they are readily available in the related literature.

G. Resource Management During Scheduling

During schedule construction, the constructor must be aware of a number of constraints on the schedule. The previous parts of this section described management of the data dependence graph during scheduling and different possible orders and techniques for scheduling the DAG. Resource management is the other major concern during scheduling. While dependences and operational latencies may allow an instruction to be scheduled in a particular cycle, the target machine may not have enough of the appropriate functional units, issue slots, or other pieces of hardware to launch or execute the instruction in the desired place. These constraints on scheduling are called *resource hazards*, and a separate module of the schedule constructor typically models all of them. This module maintains its own state in response to scheduling actions, and answers queries about whether a given instruction can be scheduled in a particular place given the already-scheduled instructions.

Early approaches accounted for resources in the underlying machine using reservation tables. A second approach used finite-state automata to model resource constraints, allowing an instruction to be scheduled in a slot if a transition existed in the resource automaton. The past decade has seen innovation in both approaches. Applying automaton theory to the FSA models, researchers have factored resource automata into simpler components, they have used reverse automata to support reverse scheduling, and they have used nondeterminism to model functional units with overlapping capabilities. Responding to these innovations in automata-based resource modeling, reservation-vector proponents have built reduced reservation vector schemes that approach the abilities and efficiencies of the new FSA techniques.

1) Resource Vectors: The basic resource vector approach involves simple accounting. A *reservation table* is a matrix with a column for each cycle of a schedule and a row for each resource in the machine. When an instruction is scheduled, the system records the resources that it uses in the appropriate table entries. Reservation tables allow easy scheduling of instructions; unscheduling can be supported by keeping a pointer from each resource to the instruction that uses it [61]. Reservation tables can easily be extended to include counted resources, where an instruction uses one resource from a hardware-managed pool of identical resources. They are less good at managing instructions that can be handled by multiple functional units (e.g., an integer add might be processed by either the ALU or the AGU). And in their simplest implementation, they require space proportional to the length of the schedule times the number of resources in the machine. Determining whether an instruction can be scheduled requires examining all of the resources used by the "template" of the instruction, which could be a large constant factor.

2) Finite-State Automata: Finite-state automata have intuitive appeal. "Can I schedule this?" is similar to "does this state machine accept?" One can view the set of resource-valid schedules as a language over the alphabet of operations. It turns out that these languages are simple enough to model using finite-state automata (FSA), one of the simplest of computational abstractions. Early models [62] built FSAs directly from the reservation vectors. Proebsting and Fraser [63] reduced the size of the FSAs by changing the underlying model. Instead of using reservation vectors as states, they abstracted to vectors that modeled whether an instruction in the current cycle would conflict with a second instruction in a later cycle.

Finite-state automata do not support backtracking, unscheduling, or cyclic scheduling well. But they could answer queries about forward, cycle scheduling (less so operational scheduling) very quickly.

3) Recent Improvements to FSAs: In their 1995 paper, Bala and Rubin [64] present a number of innovations to FSA-based resource management: factor automata, merge automata, and reverse automata. Each of them draws on well-known techniques in automata theory.

Factor automata reduce the number of FSA states (and therefore the size to store them) by observing that the different functional areas of modern machines tend to operate independently. For example, the MIPS R3000 model has separate integer and floating-point sides; they share issue and load/store hardware but not much else. These independent pieces can then be modeled by separate (factored) automata. The cross-product of the factors produces an automaton that is equivalent to the original, but the state of both factors can be represented more compactly than the state of the larger (product) automaton. The factors interact only in issue resources; otherwise they run independently. Bala and Rubin report that factoring reduced an automaton for the Alpha 21 064 of 13 524 states to two automata of 237 and 232 states.

Resource modeling across control flow merges (joins or splits) has long been a thorny engineering problem. One must model the state that results from either path. Resource vector approaches can logically OR the vectors of the parent blocks, but not until Bala and Rubin proposed join tables did the FSA approaches have a complementary technique. A join table maps from the cross-product of states to a single state, allowing two pipeline FSA states to be mapped to a single state that represents both sets of resources being used.

Adding instructions to an already-scheduled block of code has also posed engineering problems in the past. The forward automaton allows its users to verify that a sequence of instructions holds no structural hazards, but it only allows new instructions to be appended to the sequence. Insertion requires a linear rescan to verify that the new instruction did not conflict with any later instructions. Reverse automata (which is the FSA reversal of the forward automaton into a nondeterministic automaton over the power set of original automaton states) provide a partial answer to this problem, by allowing the system to model the resource constraints of those future instructions. A scheduler that maintains both forward and reverse automaton states can verify that there are no structural hazards for a single inserted instruction. However, inserting additional instructions still requires linearly rescanning to recompute the forward and reverse automaton states affected by the first inserted instruction.

What about nondeterminism? Bala and Rubin's later technical report use nondeterminism to model flexible pipeline resources that can execute a variety of instruction classes. Suppose a machine has two functional units, FU1 and FU2, and three instruction classes A, B, and C. FU1 can execute instructions of types A and B; FU2 can execute instructions of type B and C. In a deterministic automaton or a reservation vector model, scheduling a type B instruction requires it to be assigned to either FU1 or FU2. This excludes the future possibility of issuing an instruction of type A or C, respectively. With nondeterminism, the FSA can model simply issuing the type B instruction without committing it to FU1 or FU2. Then a later type A or type C instruction can still be scheduled, in effect lazily choosing the unit on which the type B instruction executes.

4) Minimal Resource Vectors: In response to Bala and Rubin's innovations, Eichenberger and Davidson [61] explored ways to build better resource vector models. Their key observation is that many explicitly-modeled resources are in fact schedule-equivalent. For example in the canonical in-order RISC five-stage pipeline, any instruction that uses the decode phase one cycle will certainly use the execute phase in the next. These stages do not require separate vector entries; they will always be allocated together. Eichenberger and Davidson present techniques to synthesize a minimal set of resources that are equivalent to but less numerous than the actual machine resources. Such reduced reservation vector models are much better users of space and time than the original vectors, although Eichenberger and Davidson do not directly compare their implementation to that of Bala and Rubin. The minimal resource vector approach supports flexible pipeline resources less elegantly than the nondeterministic FSA approach: alternate instruction locations are searched exhaustively in Eichenberger and Davidson's work.

VI. Looking Forward

ILP scheduling techniques have matured over the past 20 years. The original intuition that inspired trace scheduling and its region-scheduling descendants has blossomed into a large field from which many practical technologies have emerged. In the past, compiler writers adapted their work to match what the architects and implementers produced. Today, the requirements of all three groups contribute to designs.

A production compiler is like a bridge: it takes years to build, is used for many more years, and it requires constant repairs and improvements during its working lifetime. But bridge building and compiler construction are worlds apart in maturity and reliability. Bridge building is a well-studied engineering discipline; despite the advances of the past two decades, compiler construction remains a black art. A number of challenges remain for the research and development communities; we illustrate them by extending our bridge simile into a bridge conceit.

- *Methodology:* The past decade of architecture and optimizing compiler research benefited from a new emphasis on quantitative methods, in particular an emphasis on performance measured by (and only by) execution time. This "quantitative approach" was better than previous qualitative evaluations, but it is no more the whole picture than the load capacity of a bridge is the only salient aspect of bridge design. It is difficult to isolate the value of new compilation ideas, as they affect many pieces of a compiler in different, systemic ways. It is hard to compare techniques, as they may be embodied in very different systems. And it can be hard to publish either sort of result without reducing one's results to SpecMarks. Rather than inveighing against quantitative methods (a Luddite position), the research community should find and use more sophisticated methods that allow component-wise, isolated analysis and comparison. Bridge designers speak of tensile strength, torsional rigidity, and strength/weight; what are the equivalent metrics for optimizations and intermediate representations?
- *Infrastructure:* Designers of bridges have a large set of previous designs, each with documented histories, benefits, and drawbacks, from which to choose. Compiler researchers have relatively few, and compiler developers have even fewer because of intellectual property restrictions. This imposes many costs on the community: high entry cost for both researchers and developers, lack of comparability among designs (see previous point), inability to combine results from different projects, and a lack of standardized tools. The National Compiler Infrastructure Project [65] hoped to provide some of these benefits; the Gnu C Compiler, gcc, serves as the de-facto platform for many experiments. Both have their drawbacks.
- *Goals:* New computing realities rarely have a proper influence on compiler design. It is as if we designed all

bridges solely to maximize load, whereas some bridge designers care about cost, maintainability, stability in crosswinds, esthetics, etc. The next generation of computing devices has begun to appear, in the "embedded" space. Designers of such devices care about code size, power dissipation, heat, and unit cost. Compilers should follow suit, although few compilation systems allow such factors to be traded off.

In addition to the high-level challenges above, we list some more specific challenges to compilers and schedulers.

- *Dynamic techniques:* Post-compilation techniques, which operate at link time, load time, or run time, have become quite sophisticated. There is vast potential in combining such techniques with compile-time methods, but very little work has been done in this intersection.
- *Debugging:* Few people recognize that debugging optimized code (or DOC) is a compiler problem. Worse, debugging is a pariah like system administration: the problem that everyone must face but which is seen as too unglamorous for research. DOC can only be successful if it is part of the compiler, but the scheduling techniques described above are never presented in that light. DOC is necessary at two levels: both to verify the correct operation of optimizations, and to verify the correct operation of object code.
- *True optimization:* Ken Wilson (see Section II-B1) may have had the right idea 20 years too early. Compiling is now done on computers with enormous power; techniques that were unacceptably slow a few years ago are now practical. None of the algorithms described above scale to exploit computing power; they just run faster on newer computers. Putting it another way, developers tolerate only a fixed amount of compile time. As machines go faster, they can search more space in the same amount of time. Has the time finally arrived to reconsider compilation, solving phases exhaustively and optimally rather than heuristically? Researchers have scratched the surface of this problem, in limited areas such as register allocation [66] and code layout [67]. We have yet to see a systematic treatment of all aspects of compilation, or an approach to true optimization.

We fervently hope that the next time someone writes a survey article such as this, that some of the topics we list above will have been solved or incorporated into standard practice.

Acknowledgment

The authors thank "Reviewer A" for their insightful comments, particularly those concerning the treatment of loops in this paper.

References

[1] D. Landskov, S. Davidson, B. D. Shriver, and P. W. Mallett, "Local microcode compaction techniques," *ACM Comput. Surv.*, vol. 12, pp. 261–294, Sept. 1980.

[2] Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120b/FPS-164 family," *IEEE Comput.*, vol. 14, no. 3, pp. 18–27, 1981.

[3] J. E. Thornton, Ed., *Design of a Computer: The Control Data 6600*: Scott, Foresman and Company, Library of Congress Catalog No. 74-96 462, 1970.

[4] P. Faraboschi, G. Desoli, and J. A. Fisher, "VLIW architectures for DSP and multimedia applications—The latest word in digital and media processing," *IEEE Signal Processing Mag.*, Mar. 1998.

[5] E. G. Coffman, Jr., Ed., *Computer and Job-Shop Scheduling Theory*. New York: Wiley, 1976.

[6] J. A. Fisher, "The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources," Ph.D. dissertation, Technical Report COO-3077-161, Courant Mathematics and Computing Laboratory, New York Univ., New York, Oct. 1979.

[7] F. Astopas and K. I. Plukas, "Method of minimizing microprograms," *Automat. Contr.*, vol. 5, no. 4, pp. 10–16, 1971.

[8] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez, Jr., "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. Comput.*, vol. C-21, pp. 137–146, Feb. 1972.

[9] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Trans. Comput.*, vol. C-21, pp. 1405–1411, Dec. 1972.

[10] M. Tokoro, E. Tamura, and T. Takizuka, "Optimization of microprograms," *IEEE Trans. Comput.*, vol. C-30, pp. 491–504, July 1981.

[11] D. Jacobs, J. Prins, P. Siegel, and K. Wilson, "Monte Carlo techniques in code optimization," in *Proc. 15th Annu. Workshop Microprogramming*, Oct. 1982, pp. 143–148.

[12] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *14th Annu. Microprogramming Workshop (MICRO-14)*, 1981, pp. 183–198.

[13] M. Lam, "Sofware pipelining: An effective scheduling technique for VLIW machines," in *Proc. SIGPLAN'88 Conf. Prog. Language Design and Implementation*, Atlanta, GA, 1988, pp. 318–328.

[14] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Annu. Int. Symp. Microarchitecture*, Nov. 1994, pp. 63–74.

[15] J. A. Fisher, D. Landskov, and B. D. Shriver, "Microcode compaction: Looking backward and looking forward," in *Proc. National Computer Conf.*: AFIPS, 1981, pp. 95–102.

[16] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *SIGPLAN'88 Conf. Programming Language Design and Implementation*, pp. 308–317.

[17] ——, "Perfect pipelining: A new loop parallelization technique," in *Proc. 2nd Eur Symp. Programming*, ser. Lecture Notes in Computer Science: Springer-Verlag, Mar. 1988, vol. 300, pp. 221–235.

[18] J. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. 30, pp. 478–490, July 1981.

[19] J. R. Ellis, "Bulldog: A compiler for VLIW architectures," Dept. Computer Science, Yale Univ., Tech. Rep. YALEU/DCS/RR-364, Feb. 1985.

[20] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Three superblock scheduling models for superscalar and superpipelined processors," Center for Reliable and High-Performance Computing, Univ. Illinois at Urbana-Champaign, Rep. CRHC-91-25, Oct. 1991.

[21] W. A. Havanki, "Treegion scheduling for VLIW processors," M.S. thesis, Dept. Electrical and Computer Engineering, North Carolina State Univ., Raleigh, NC, 1997.

[22] J. A. Fisher, "Global code generation for instruction-level parallelism: Trace scheduling-2,", HP Laboratories Tech. Rep. HPL-932-43, 1993.

[23] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its uses in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, July 1987.

[24] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation-based scheduling," in *Proc. ACM/IEEE Design Automation Conf.*, 1990, pp. 444–449.

[25] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: An introduction and motivation," in *Proc. 28th Annu. Int. Symp. Microarchitecture*, Dec. 1995, pp. 158–168.

[26] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. O. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *J. Supercomput.*, vol. 7, pp. 229–248, Mar. 1993.

[27] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," in *Proc. 1994 Conf. Programming Language Design and Implementation (PLDI)*, June 1994, pp. 196–205.

[28] Intel. VTune: Visual Tuning Environment (1997). [Online]. Available: http://developer.intel.com/design/perftool/vtune/index.htm.

[29] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System support for automatic profiling and optimization," in *Proc. 16th ACM Symp. Operating Systems Principles, USENIX*, Oct. 1997, See also [Online]. Available: http://www.eecs.harvard.edu/morph, to be published.

[30] J. M. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. A. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?," Digital Equipment Corporation Systems Research Center, Palo Alto, CA, Tech. Note 1997-016, July 1997.

[31] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proc. ACM SIGPLAN 2000 Conf. Programming Language Design and Implementation*, Vancouver, BC, Canada, 2000.

[32] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," in *Conf. Record 19th ACM Symp. Principles of Programming Languages*, Jan. 1992, pp. 59–70.

[33] ——, "Efficient path profiling," in *Proc. Micro 96*, Dec. 1996, pp. 46–57.

[34] J. Larus, "Whole Program Paths," in *PLDI '99*, May 1999.

[35] A. Young, "Path-based compilation," Ph.D. dissertation, Harvard Univ., Cambridge, MA, Oct. 1997.

[36] T. Ball and J. Larus, "Branch prediction for free," in *Proc. SIGPLAN '93 Conf. Programming Language Design and Implementation*, June 1993, pp. 300–313.

[37] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *27th Int. Symp. Microarchitecture*. San Jose, CA: IEEE, Nov. 1994, pp. 1–11.

[38] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 1, 1997.

[39] SPEC. SPEC CPU95, Version 1.0. (1995, Aug.). Standard Performance Evaluation Corporation. [Online]. Available: http://www.specbench/org.

[40] S. S. Muchnik, *Advanced Compiler Design & Implementation*. San Mateo, CA: Morgan Kaufmann, 1997.

[41] R. Morgan, *Building an Optimizing Compiler*. Boston, MA: Digital Press, 1998.

[42] T. Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proc. 24th Annu. Int. Symp. Microarchitecture*, Nov. 1991, pp. 51–61.

[43] S. Pan, K. So, and J. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *ASPLOS-V*, Oct. 1992, pp. 76–84.

[44] C. Young, N. Gloy, and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction," in *Proc. 22nd Annu. Int. Symp. Computer Architecture*, Santa Margherita Ligure, Italy, June 22–24, 1995, pp. 276–286.

[45] T. Ball and J. Larus, "Programs follow paths,", Tech. Rep. MSR-TR-99-01, Jan. 1999.

[46] Z. Wang, K. Pierce, and S. McFarling, "BMAT—A binary matching tool," in *Proc. Feedback Directed Optimization 2*, 1999.

[47] G. Albert, "A transparent method for correlating profiles with source programs," in *Proc. Feedback Directed Optimization 2*, 1999.

[48] J. Fisher and S. Freudenberger, "Predicting conditional branches from previous runs of a program," in *Proc. 5th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1992, pp. 85–95.

[49] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proc. 2nd Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Apr. 1987, pp. 180–192.

[50] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. O'Donnell, and J. C. Ruttenberg, "The multiflow trace scheduling compiler," *J. Supercomput.*, vol. 7, pp. 51–142, Mar. 1993.

[51] S. M. Moon and K. Ebcioglu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW Processors," in *Proc. MICRO-25*: IEEE Press, Dec. 1992, pp. 55–71.

[52] W. A. Havanki, S. Banerjia, and T. M. Conte, "Treegion scheduling for wide issue processors," in *Proc. 4th Int. Symp. High-Performance Computer Architecture*, Feb. 1998, pp. 266–276.

[53] S. A. Mahlke *et al.*, "Effective compiler support for predicated execution using the hyperblock," in *Proc. 25th Int. Symp. Microarchitecture (MICRO 25)*, 1992, pp. 45–54.

[54] C. Young and M. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *ASPLOS-VI*, Oct. 1994, pp. 232–241.

[55] M. Schlansker and V. Kathail, "Critical path reduction for scalar programs," in *Proc. 28th Annu. Int. Symp. Microarchitecture*, Ann Arbor, MI, Nov. 29–Dec. 1, 1995, pp. 57–69.

[56] C. Chekuri, R. Johnson, R. Motwani, B. K. Natarajan, B. R. Rau, and M. Schlansker, "Profile-driven instruction level parallel scheduling with applications to super blocks," in *Proc. 29th Annu. Int. Symp. Microarchitecture (MICRO-29)*, Paris, France, Dec. 2–4, 1996.

[57] P. Faraboschi, G. Desoli, and J. A. Fisher, "Clustered instruction-level parallel processors," Hewlett-Packard, Tech. Rep. HPL-98 204, 1998.

[58] S. M. Freudenberger and J. C. Ruttenberg, "Phase ordering of register allocation and instruction scheduling," in *Code Generation—Concepts, Tools, Techniques: Proc. Int. Workshop Code Generation*, May 1992.

[59] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen, "Combining register allocation and instruction scheduling technical report," Courant Institute, Tech. Rep. 698, July 1995.

[60] Norris and L. L. Pollock, "Register allocation sensitive region scheduling," in *PACT '95: Int. Conf. Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995.

[61] A. E. Eichenberger and E. S. Davidson, "A reduced multipipeline machine description that preserves scheduling constraints," in *PLDI 96*.

[62] E. S. Davidson, A. T. Thomas, L. E. Shar, and J. H. Patel, "Effective control for pipelined processors," in *Proc. COMPCON75*: IEEE, Mar. 1975, pp. 181–184.

[63] T. A. Proebsting and C. W. Fraser, "Detecting pipeline structural hazards quickly," in *21st Annu. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Jan. 1994, pp. 280–286.

[64] V. Bala and N. Rubin, "Efficient instruction scheduling using finite state automata," in *Proc. MICRO-28*.

[65] The National Compiler Infrastructure Project home page [Online]. Available: http://www.cs.virginia.edu/nci/.

[66] D. W. Goodwin and K. D. Wilken, "Optimal and near-optimal global register allocation using 0–1 integer programming," *Softw.—Pract. Exp.*, vol. 26, pp. 929–965, Aug. 1996.

[67] C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith, "Near-optimal intraprocedural branch alignment," in *Proc. ACM SIGPLAN 97 Conf. Prog. Lang. Design and Implementation*. New York: ACM, June 1997.

Paolo Faraboschi (Member, IEEE) received the Ph.D. degree in electrical engineering and computer science from the University of Genova, Italy, in 1994.

He is a Principal Research Scientist at Hewlett-Packard Laboratories, Cambridge, MA. His interests include VLIW architectures and compilers, instruction level parallelism, and embedded computing.

Joseph A. Fisher (Senior Member, IEEE) received the Ph.D. degree from New York University in 1979.

He is a Hewlett-Packard Fellow and directs the Hewlett-Packard Laboratories research lab, Cambridge, MA. He is best known for his contributions to VLIW Architectures and compiling for Instruction-Level Parallel architectures and has worked in high-performance embedded processing since 1994. Prior to joining HP, he was on the Computer Science faculty of Yale University and cofounded Multiflow Computer, which manufactured high-performance VLIW processors.

Dr. Fisher won an NSF Presidential Young Investigator Award in 1984, and was Eli Whitney Connecticut Entrepreneur of the Year in 1987.

Cliff Young (Member, IEEE) received the Ph.D. degree in computer science from Harvard University, Cambridge, MA, in 1997.

He works in the Computing Sciences Research Laboratory at the Computing Concepts Department, Bell Laboratories, Murray Hill, NJ. Most of his graduate work involved playing tricks to make computers go faster: improving performance through compiler optimizations driven by exotic statistics and applying architectural techniques such as branch prediction. Since joining Bell Labs, his research interests have broadened to include multicomputers (i.e., big servers), information theory (i.e., even more exotic statistics), and distributed systems (in particular, handheld, wireless-connected devices).