TEMPLATE BUNDLING FOR EPIC ARCHITECTURES

BY

SAIN-ZEE UENG

B.S., University of California at Berkeley, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

ABSTRACT

One of the differences between out-of-order and in-order computer architectures is the dispersal of operations to functional units. This responsibility is part of the transfer of power from hardware (out-of-order) to the compiler (in-order). As a result of this shift, the hardware support for operation dispersal was simplified. IPF, the first hardware implementation of an EPIC in-order architecture, introduced new concepts to further simplify the hardware's operation dispersal. However, they present additional constraints to the compiler's scheduler. This thesis presents the template bundling algorithm to extend the IMPACT compiler to handle these new constraints. An exhaustive, systematic exploration over the newly introduced search space is employed to produce a schedule that conforms to the scheduler's performance expectations while keeping compilation time under control via efficient implementation.

ACKNOWLEDGMENTS

I would first like to thank my adviser, Professor Wen-mei Hwu, for his guidance and support. Many thanks go out to the IMPACT research group members, both past and present, who have contributed to this framework which my work is built upon. Special thanks are due to John Sias, without whose assistance, patience, and insight this work would not have been possible, and Marie Conte, who believed in me and gave me the chance. I would like to thank my parents, my sister, and my family for their support and encouragement. Last, but not least, I would like to thank Dr. Hungwen Li for motivating me towards my graduate studies.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Out-of-order computer architectures employ hardware resources to control the dispatch of operations to functional units. Many modern-day architectures, such as superscalar machines, operate under this framework. This requires the hardware to support dynamic, near-arbitrary dispersal of operations to functional units in the presence of scheduling concerns such as dependences. In-order architectures like Very Long Instruction Word (VLIW), in contrast, rely upon the compiler to schedule operations statically for execution in the hardware. In particular, the compiler's scheduler needs to ensure that the final code obeys many constraints, such as dependences, latencies, resource usage, etc.; otherwise, performance may suffer, or worse yet, results may be incorrect. This, however allows the processor's issue logic to be much simpler. One variation of in-order machines, the Explicitly Parallel Instruction Computing (EPIC) architecture, gives the compiler even greater responsibilities and power when compiling and scheduling code. In one EPIC architecture, the Intel Itanium Processor Family (IPF) [1], new concepts are introduced to the instruction format to simplify the dispersal of instructions in hardware. Unfortunately, these same specifications significantly complicate the compiler's job during scheduling. This thesis will describe the augmentations of a compiler to handle these new constraints.

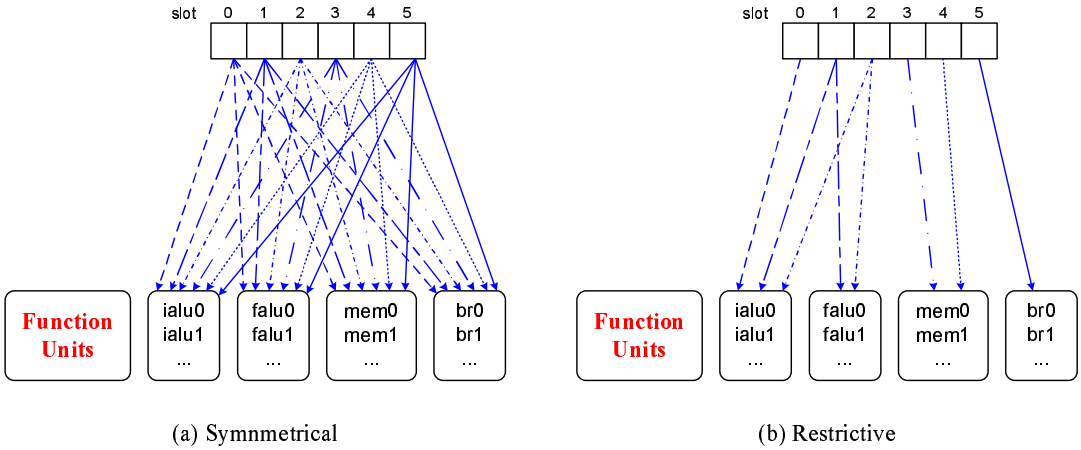


Figure 1.1 Generic EPIC operation dispersal mind-sets

EPIC architectures generally assume one of two models for dispersing operations from slots to functional units: (a) symmetrical and (b) restrictive. Figure 1.1(a) shows the symmetrical dispersal assumption. Any slot can dispatch an operation to any functional unit each cycle. The blue lines indicate how each slot in a six-issue machine can reach any functional unit. As long as there is an appropriate functional unit available, the operation will execute as scheduled. Figure 1.1(b) shows the restrictive dispersal assumption. Each slot can disperse its operation to only a restricted set of functional units, governed by a fixed mapping of slots to functional units. This mapping in turn restricts the operations that can scheduled in each slot. Variations between these two extremes also exist.

IPF groups operations into bundles, each of which contains three slots. Each implementation is capable of processing some number of bundles per cycle in an issue group. For example, the current implementations of IPF are designed to process two bundles per cycle, making them nominally six-issue machines. IPF classifies the different types of instructions in the instruction

set architecture (ISA) like memory, integer, floating point, and branch instructions into syllable types. When encoding operations, IPF requires bundles to be generated according to templates that assign a syllable type to each slot of the bundle. Syllable types in turn restrict what operations can be scheduled in that slot and what functional units are reachable from that slot. Finally, dispersal rules determine what functional unit an operation ends up in the hardware given its instruction type, syllable type, slot position, and preceding operations in the cycle. Rather than require that every functional unit be reachable by every slot or restrictively define which functional units a slot is able to disperse to, the set of functional units reachable now varies from slot to slot and cycle to cycle according to the templates and syllables specified.

Figure 1.2 demonstrates how these new restrictions in IPF complicate the scheduler's job compared to the generic EPIC architecture. The set of functional units reachable from each slot is determined by the syllables of the template, as shown by the orange letters and dashed lines. Certain instructions in IPF are even more restrictive, allowing their execution on an even smaller subset of functional units. The integer load instruction in slot 3, with a legal subset of only the M0 and M1 functional units (the dashed blue line and circle), is used here as an example. Finally, according to the dispersal rules, the dispatch of operations to functional units in IPF is affected by preceding operations in the cycle. The operation in slot zero is a floating point load that may issue to M0 through M3, as indicated by the dashed grey line and circle. The dispersal rules dictate that it will execute on the lowest numbered legal M-functional unit currently available, M0 in this case (red circle and line). Therefore, the integer load operation must disperse to M1. If the operation in slot 1 was also a floating point load, it would have

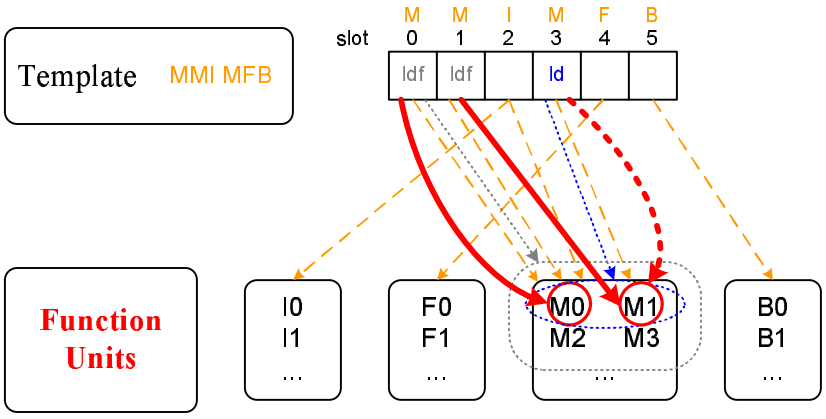


Figure 1.2 IPF operation dispersal mind-set

taken M1, the lowest numbered functional unit out of M0 through M3 still available, leaving no functional unit for the integer load operation in slot 3, which is why it has a dashed red line instead. However, if the integer load was in a slot that preceded all of the floating point load operations, no problems would exist until the machine runs out of M-functional units. For the generic EPIC architecture, functional units are assigned to operations in slots on a first-come-first-served basis; the operation scheduled first by the scheduling algorithm will have first choice. For IPF, the functional unit assigned to an operation depends on those in preceding slots, independent of the order the scheduling algorithm picks the operations. The scheduler of EPIC compiler would need to be changed to adapt to these new restrictions of IPF.

The work for this thesis is based on the IMPACT research compiler, developed by the IMPACT research group at the University of Illinois at Urbana-Champaign. The call stack of the existing IMPACT compiler's scheduler is shown under the "old path" of Figure 1.3. A scheduling algorithm, cyclic or acyclic, decides the cycle time and slot range for an operation based

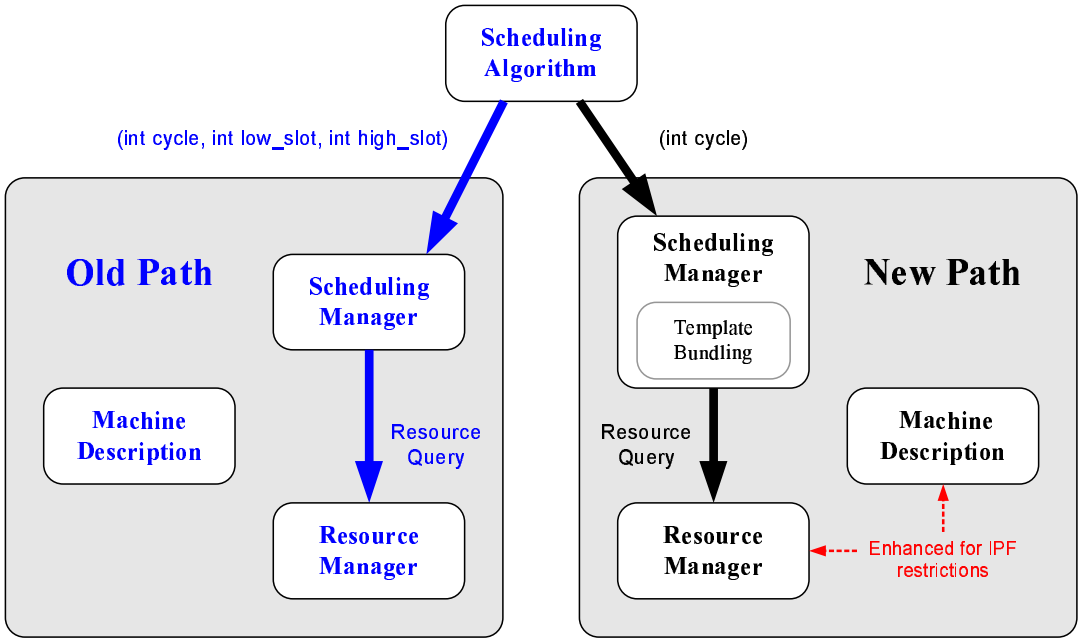


Figure 1.3 IMPACT scheduler call stack

on its dependencies and passes the information to the scheduling manager. The scheduling manager calls the resource manager to determine if an appropriate slot and functional unit are available. If the answer is yes, those resources are marked as taken, and the scheduling manager updates the status of the operation and the operations it affects. The resource manager derives its machine-specific information from the machine description. The various duties of the scheduler (timing, placement, resource description and management, etc.) are thus spread out amongst specialized components.

The scheduler must consider all of the new factors introduced by IPF (bundles, templates, syllables, and dispersal rules) in addition to traditional constraints like dependences and come up with the best-performing schedule possible. The “new path” of Figure 1.3 outlines the new

model dealing with the IPF constraints. The scheduling algorithms are not altered in any way, though their interface to the scheduling manager changes slightly, allowing the compiler to maintain its modular separation of scheduling responsibilities. The scheduling algorithm still directs the scheduling manager with the intended cycle for an operation, but the scheduling manager is now responsible for finding a combination of templates and a permutation of operations within the cycle such that all of the cycle's operations are assigned to slots that allow for simultaneous issue, given the specified constraints. The scheduling manager still relies upon the resource manager to determine if the appropriate resources are available. However, the resource manager has been enhanced as necessary to accommodate the new restrictions. The constraints themselves are described in the machine description so that they can be specified per machine implementation and be accessed by different parts of the compiler.

This thesis proposes a framework at the scheduling manager level, the *template bundling algorithm*, for IPF instruction scheduling that takes these new constraints into consideration. The machine description and resource manager are augmented accordingly to support this effort. In Chapter 2, the new restrictions introduced by IPF are defined in detail, and the problems they cause are analyzed. Chapter 3 presents the implementation of the template bundling algorithm and the associated support and steps through the algorithm to demonstrate how it handles the various constraints and avoids the potential problems. The complexity of the template bundling algorithm is also analyzed. Numerical results are presented in Chapter 4, related work in Chapter 5, and future work in Chapter 6.

CHAPTER 2

BACKGROUND

This chapter first goes over the issue and dispersal structures of generic EPIC architectures. The discussion will be based primarily on the features assumed by the IMPACT compiler. It then introduces the constraints and features new to IPF. An in-depth analysis then details the constraints and their effects upon the schedule and scheduling process. The discussion will focus on the Itanium 2 implementation of IPF. An example sequence of operations will be employed to help concretize the concepts and the interplay among them.

2.1 Generic EPIC Architecture

EPIC architectures are in-order machines, meaning the compiler is responsible for producing a schedule of the operations for the hardware to execute. Due of this requirement, the compiler must be aware of the hardware resources available. The IMPACT compiler uses a *machine description* [2] to describe the capabilities of the target hardware and its interaction with the instructions in the ISA. This section describes the machine model typically used by the IMPACT compiler, an amalgam of the two generic EPIC operation dispersal mind-sets presented earlier.

An EPIC architecture able to dispatch n operations in a single cycle is referred to as an n-issue machines. Figure 2.1 shows a six-issue machine. Each box represents a *slot* for an

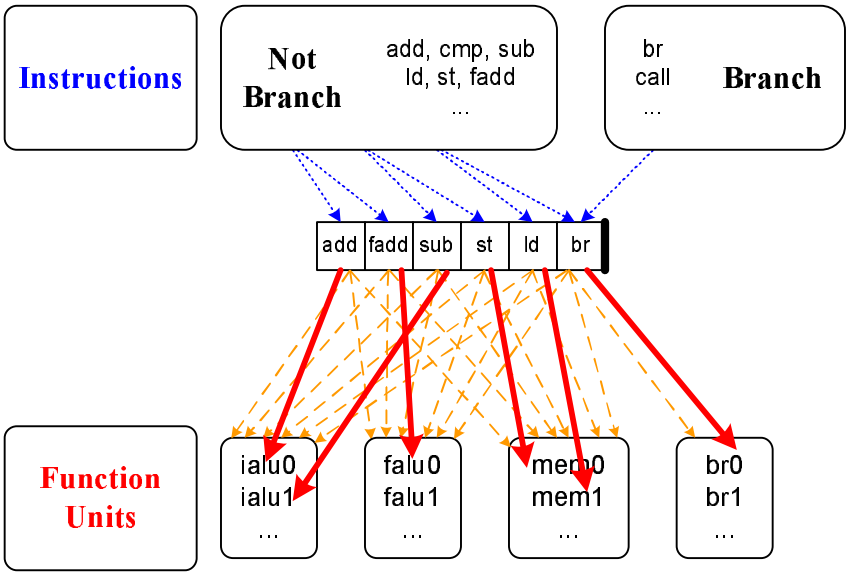


Figure 2.1 Generic EPIC architecture

operation. Since all of the operations are to be dispersed in the same cycle, there is a *cycle boundary*, indicated by the thick black bar, at the end of the issue. The machine model used by the IMPACT compiler resembles the symmetrical operation dispersal model for all but the last slot. The machine is assumed to issue only one branch instruction per cycle, and it must be the last instruction. This is shown by the single dashed blue line extending from the set of branch instructions to the last issue slot. (IPF has different restrictions on the number and placement of branches in a cycle, which will be discussed in Section 2.3.) Nonbranch instructions may be placed in any of the slots, shown by the multiple dashed blue lines connecting the set of nonbranch instructions to all of the slots. There are also minimal restrictions as to the functional unit to which a slot can disperse. The last slot can disperse to any functional unit, whereas the remaining slots can disperse to any functional units except the branch functional unit. This is

shown by the dashed orange lines connecting the slots to the functional units. The functional units consuming the operations scheduled are indicated by the solid red lines. Generic EPIC architectures are not concerned with the specific functional unit in which each operation ends up and do not specify any kind of set mapping.

2.2 Overview of IPF Restrictions

The additional constraints introduced by IPF represent a continual evolution of operation dispersal for in-order architectures. The desired effect of these restrictions is to reduce the complexity of hardware issue logic so that it does not have to provide the number of paths required for all functional units to be reachable from all slots at all times. The more limited dispersal of operations is configurable via templates and syllables on a cycle by cycle, bundle by bundle, and slot by slot basis. This places additional burdens on the compiler and its scheduler.

IPF groups and encodes its operations in *bundles*, each containing three slots. Every implementation of IPF can operate on a certain number of bundles per cycle. Both Itanium and Itanium 2 operate on up to two bundles per cycle, so one can essentially think of them as 6-issue machines. IPF specifies cycle boundaries by *stop bits*. Stop bits can be placed at the end of any bundle and within bundles in some cases. Some of the possible placements of stop bits are indicated by the grey bars and the actual cycle boundary stop bit by the black bar in Figure 2.2. Stop bits are prescribed in the *template* of each bundle. Every bundle must correspond to a valid template, which contains information about the stop bits in that bundle and an assignment of *syllable types* to slots. Figure 2.2 shows in orange lettering the bundles being

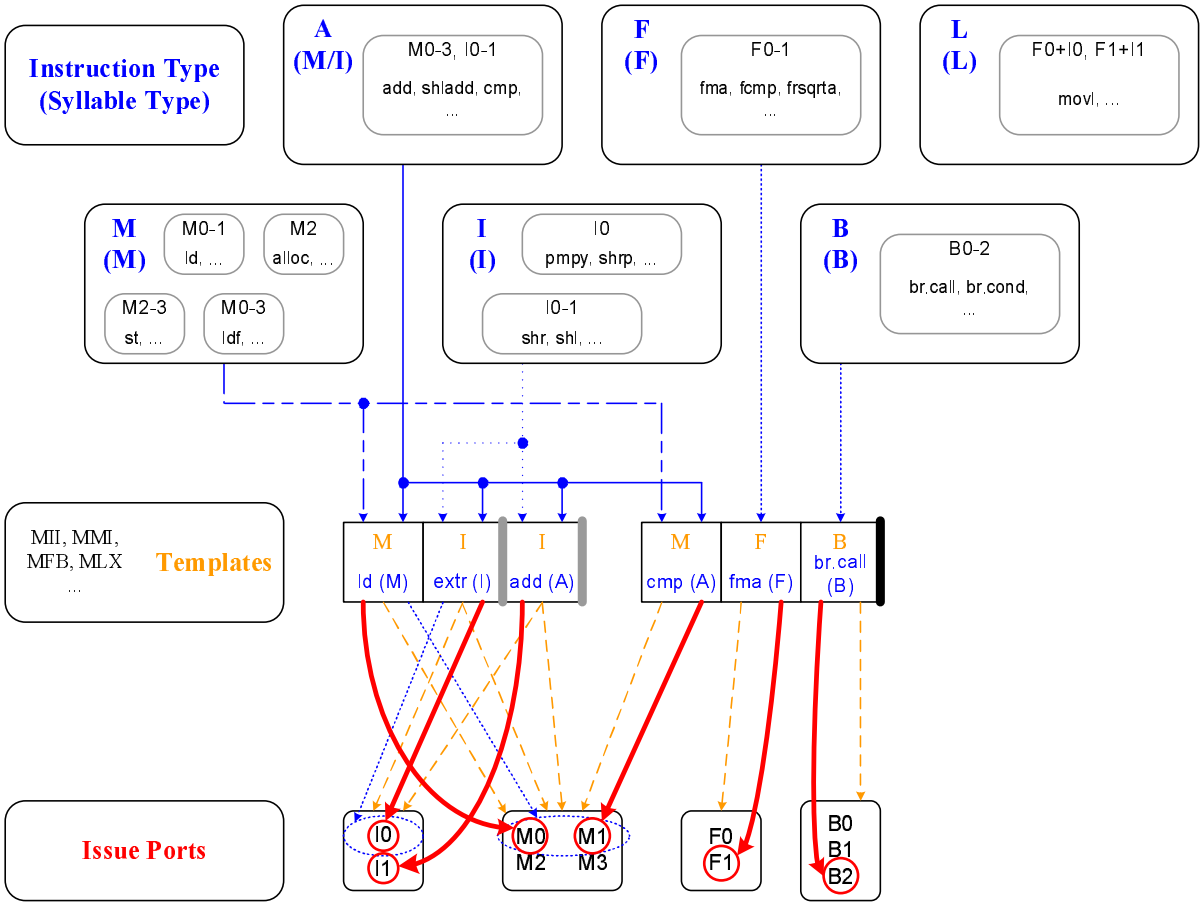


Figure 2.2 IPF restrictions overview

assigned templates out of the set of valid templates and the syllables being mapped into the slots of the bundles. The template information of each bundle is passed to the hardware along with the operations contained within it. The syllables assigned to the slots determine what *issue ports* can be reached from that slot, shown in Figure 2.2 by the dashed orange lines. Issue ports are an intermediate step between how operations are distributed from the slot to the functional units in IPF; each operation is assigned to the appropriate functional unit through an issue port. It is the lowest level of machine modeling information needed by the compiler.

Every instruction in the ISA of IPF belongs to an *instruction type*. An instruction can only be scheduled in a slot that contains a syllable compatible with its instruction type. Certain instruction types are more flexible than others and are compatible with more than one syllable type. These instruction types are said to have a *superset syllable*. Figure 2.2 shows the different instruction types in IPF (boxes at the top with a capital letter in blue), the set of syllables the instruction types are compatible with (letters in blue in parentheses), and a sample of instructions of each type. The blue lines from the instruction types to the slots indicate the slots an instruction type can go to in an MII MFB template combination. Notice how the A-instruction type, which has a superset syllable, can be assigned to either the M or I-syllable slots.

Not all Itanium 2 functional units can handle all of the instructions of the corresponding instruction type. The instructions in Figure 2.2 are further separated within each instruction type according to the set of issue ports that can handle that particular instruction. For example, integer load instructions (`ld`) belong to the M-instruction type and can be dispersed to either the M0 or M1-issue port. Dashed blue lines and circles indicate the set of issue ports allowed for the operation scheduled in the slots for restrictive operations, like the integer load instruction scheduled in the first slot. Given the slot, the syllable of the slot, the restrictions of the operation in the slot, and the preceding operations in the cycle, *dispersal rules* determine the particular issue port that the operation is dispersed to, as indicated by the solid red lines. The fact that previous operations directly affect the dispersal of the current operation is an important distinction between IPF and generic EPIC architectures. The previous scheduler flow for the IMPACT compiler was unable to handle this.

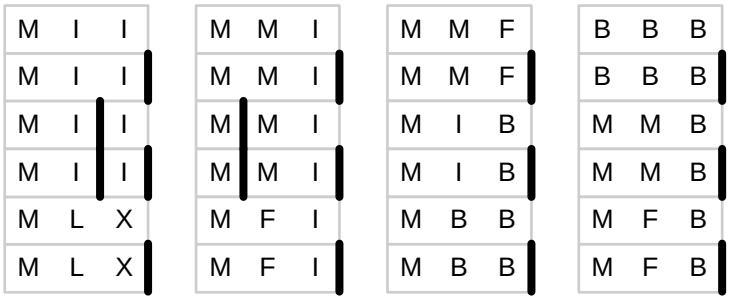


Figure 2.3 Valid templates for IPF

2.3 In-Depth Analysis of IPF Restrictions

This section details the constraints introduced by IPF as implemented on the Itanium 2. The new terms introduced in Section 2.2 will be explained here in detail. The discussion first focuses on the various templates and their formation, and then turns towards the dispersal of operations.

2.3.1 Templates

Every bundle in IPF, each of which contains three slots, must conform to a template. Templates contain information about the syllables and stop bits of the bundle and are passed to the hardware along with the operations scheduled in the bundle. There are five syllable types in IPF: M, I, F, B, and L. Only certain permutations of syllables are considered valid templates. The current set of valid templates for IPF is shown in Figure 2.3. There is room in the IPF specification for eight new templates to be defined in the future.

Each syllable takes up one or two slots in the bundle. The L-syllable consumes two slots, so only two instructions can be scheduled in the MLX template. (X is a place holder rather than another syllable.) Each syllable must be assigned a compatible operation. The operation's instruction type determines its compatibility. The operation assigned to the syllable can be a *NOP* operation if there is no operation available. NOP operations in IPF do not affect the machine state and correctness of the program; however, they do consume resources and affect the dispersal of operations. Therefore, NOP operations in IPF need to be carefully considered and scheduled, unlike the benign NOP operations previously assumed by the IMPACT compiler.

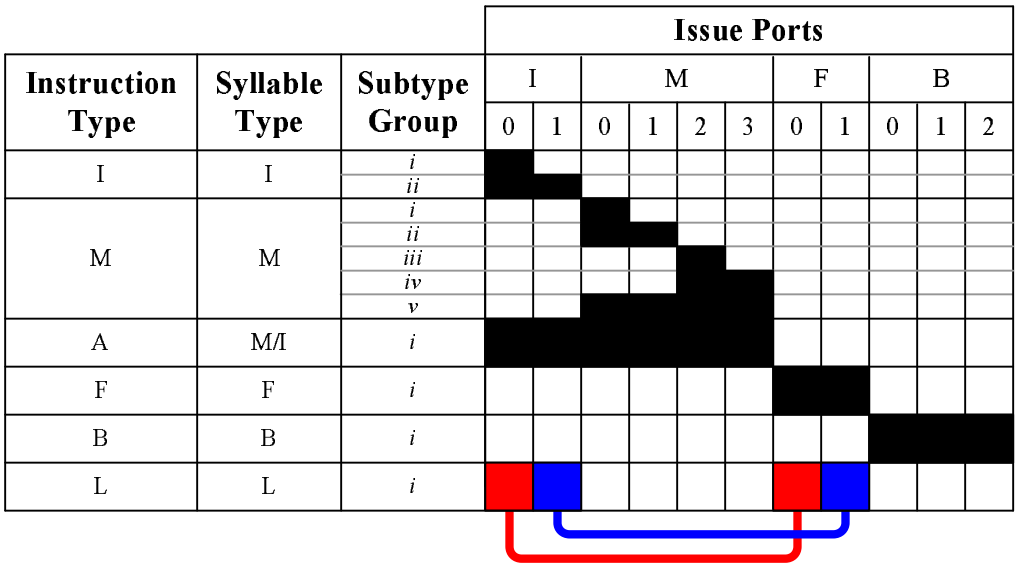
Explicit stop bits denote cycle boundaries in IPF templates, represented by the thick black bars in Figure 2.3. Note that for every permutation of syllables that forms a valid template, a version exists with a stop bit at the end of the bundle. There are also two templates that can have a stop bit in the middle of the bundle, specifically MII and MMI. They were referred to as *compressed templates* in [3], and this thesis will follow that terminology. Implicit cycle boundaries called *split issues* may also occur at locations not indicated by explicit stop bits (see Section 2.4).

Intuitively, IPF may be viewed as a constrained EPIC machine whose paths from instruction slots to functional units are reconfigurable via template specifications. Compressed templates and explicit stop bits even allow IPF machines to appear like anything between a one-issue to a six-issue machine in certain cycles. The compiler's scheduler now has the power and responsibility of partially reconfiguring the dispatch hardware through template selection and stop bit insertions to maximize performance and code density.

2.3.2 Dispersal

Every instruction in the ISA of IPF belongs to an instruction type. These major instruction types represent groups of instructions that can be assigned to the same set of syllables. Figure 2.4 shows the six major instruction types and the set of compatible syllable types. Certain instruction types have superset syllables compatible with multiple syllable types. The A-instruction type in IPF is one such example, being compatible with both the M and I-syllable types. Within each instruction type there are also subtypes. Subtypes further subdivide the instructions by different characteristics, such as instructions with different operand types like architecture registers, immediates, and memory location, or different memory operations like loads versus stores. Every instruction in the IPF ISA belongs to one and only one subtype. All of the instructions in the same subtype usually have the same issue port constraints, a subset of those associated with the major instruction type.[1] In other words, the set of valid issue ports for an instruction is determined by what instruction subtype it belongs to, as shown in Figure 2.4. Although there are many instruction subtypes, many of them share the same characteristics with regard to issue port constraints. Consequently, they are grouped into *subtype groups* in the figure. The assignment of valid issue ports to instruction subtypes is implementation specific. This is how hardware asymmetry of issue ports and functional units is expressed in IPF. For example, memory instructions belong to the M-instruction type. However, integer load instructions belong to a subtype group (*ii*) that can only disperse to the M0 or M1-issue port and integer stores belong to a different subtype group (*iv*) that can only disperse to the M2 or

[1]There are some subtypes in Itanium 2 (M6, M7, M8, and M24) where instructions have nonuniform issue port constraints.



Note: L-instructions require two issue ports per operation; red or blue set.

Figure 2.4 Mapping of instruction types to syllable types and issue ports on the Itanium 2

M3-issue port, whereas floating point loads belong to a subtype group (*v*) that can disperse to any of the M-issue ports, M0, M1, M2, or M3. Note that L-instruction type instructions in IPF take two issue ports per operation, either F0 and I0 or F1 and I1. In the interest of brevity, the large set of compatible issue ports for A-instruction type operations will be denoted as simply A.

Every operation, including each NOP operation, is mapped from the syllable and slot to a functional unit through an issue port. On Itanium 2 there are 11 issue ports: M0, M1, M2, M3, I0, I1, F0, F1, B0, B1, and B2. Issue ports present an intermediate layer between the syllables and functional units and is the lowest level of resource management that the compiler needs to model. The dispersal rules map the operation assigned to a syllable to issue ports according to the following parameters: (a) the slot number, (b) the syllable of the slot, (c) the

instruction subtype of the operation, and (d) what has preceded the operation in the current cycle. Each implementation of IPF defines its own set of dispersal rules that the compiler must take into consideration when scheduling. A simple example in Itanium 2 is the dispersal rule for slots with a B-syllable. If the slot is in the first position of the bundle the operation will be dispersed to B0, the second position goes to B1, and the third position goes to B2. Viewing the architecture as a 6-issue EPIC machine with the slots numbered from 0 to 5, an operation in slot 0 or 3 with a B-syllable would disperse to B0, slot 1 or 4 would go to B1, etc. A more complicated example would be an integer store instruction. The instruction type says integer store instructions must be in an M-syllable slot and the subtype restricts the dispersal to either the M2 or M3-issue port. The dispersal rule assigns the operation to the lowest issue port first, i.e. the operation will be dispersed to M2 if possible. If both M2 and M3-issue ports are occupied, then the integer store operation cannot be scheduled in this cycle without detrimental effects (see split issue in Section 2.4), even if other M-issue ports, such as M0 and M1, are available. Superset syllables complicate the dispersal rules because the subset of issue ports that are legal is much larger. On Itanium 2, an A-instruction type operation assigned to an M-syllable slot can be dispersed to either the M0, M1, M2, or M3 issue port. However, if that same instruction is assigned to an I-syllable slot, it will then be dispersed to either the I0, I1, M0, M1, M2, or M3 issue port. The order is very important; the architecture will only disperse to a M-issue port if both I-issue ports are occupied by other operations scheduled in slots before the current one. This flexibility of dispersal for superset syllables was not available

in the Itanium implementation. On Itanium, an A-instruction type operation assigned to an I-syllable slot will disperse to either the I0 or I1-issue port only. Although the greater flexibility of superset syllables complicates scheduling on Itanium 2, it also enables scheduling for better performance and code compactness. Our template bundling algorithm handles the additional flexibility of Itanium 2, but can be easily scaled back to the Itanium model. Please refer to [4] for the full set of dispersal rules for Itanium 2.

With the introduction of dispersal rules, the symmetrical view of operation dispersal to functional units by generic EPIC architectures no longer applies. Instead of only making sure that resources are available, the order in which operations are scheduled within a cycle also needs to be considered now. For example, the `extr` instruction in Itanium 2 can only be executed via the I0-issue port. The instruction `shr` is an I-instruction type instruction that can be executed by either the I0 or I1-issue port. From a generic EPIC architecture's point of view, the order in which these two operations are scheduled in a single cycle does not matter. It assumes that `extr` will be executed by the I0-issue port. As long as no other operation in the cycle requires an I-issue port, the scheduler for the generic EPIC architecture would have done its job. The dispersal rules of IPF, on the other hand, dictates that the lower numbered issue port of the set of legal issue ports will be used first. This means that if `shr` is scheduled in an earlier slot than `extr`, then a split issue (Section 2.4) will occur before `extr` since the I0-issue port would have already been taken by `shr`. If the order is reversed, the operations would run as scheduled. Because of these new constraints, the scheduler must be enhanced beyond what is required for the generic EPIC architecture.

2.4 Legality and Performance

If issue ports and dispersal rules are not properly taken into consideration by the compiler's scheduler, *split issue* may occur. In essence, split issues are cycle boundaries that occur without an explicit stop bit. Split issue can occur due to *resource oversubscription* [5]. Since Itanium had only two M-issue ports and two I-issue ports, template combinations such as MMI MMI or MII MII would result in split issue due to resource oversubscription of M or I-issue ports. With the added M-issue ports and additional flexibility of the superset syllable A, Itanium 2 should be able to avoid resource oversubscription unless the template selection was poor or the dispersal rules were ignored. However, certain templates and template combinations can still cause split issue on Itanium 2 [4]. A split issue always occurs after either a BBB or MBB bundle. Split issue can also happen after a MIB, MFB, or MMB bundle unless the B-syllable slot contains an NOP operation. Resource oversubscription can also still occur, e.g., MIB BBB, or MII MMI with I-instruction type operations filling all of the I-syllable slots. Note that in the case of the MII MMI combination, resource oversubscription would not happen on Itanium 2 if the last I-syllable is occupied by an A-instruction type operation, thanks to the added flexibility of superset syllable operations. However, although split issues cause performance loss by introducing cycle boundaries not planned by the compiler's scheduler, the code is still accepted by the hardware and generates the correct output. This raises the issue of legal, or acceptable, code for IPF.

The following concepts introduced in IPF: bundles, templates, and syllables, must be obeyed in order for the scheduled code to be considered legal and to be accepted by the assembler. If these specifications are not obeyed, the code generated by the compiler will not get past the assembler because it will not run on the hardware. On the other hand, keeping track of how operations are dispersed to issue ports and what resources are actually used in a cycle is a performance issue. The compiler generated code should be accepted by the assembler and the hardware and the output should be correct even if these issues were ignored (provided that the code is legal). However the code may not perform as the compiler anticipated and scheduled for.

Table 2.1 presents an example of operations that will be scheduled, including the instruction type, subtype group, syllable type, and issue port of each operation. This sequence of instructions was conceived to demonstrate as many of the pitfalls of scheduling with the new restrictions as possible. Assume that none of the memory addresses will ever alias with each other, and that the compiler knows this. The dependences (blue) and antidependences (red) between the operations are drawn out. As in IPF, antidependences are allowed in the same cycle as long as the read precedes the write. NOP operations disperse, consume resources, and affect the dispersal of other operations. The various legal schedules that can be formed out of these operations and the performance that can actually be expected when run on hardware will be examined.

Figure 2.5 shows eight different legal schedules of the operations presented in Table 2.1. Every bundle is labeled with its template to the left, and the explicit stop bits are shown by

Table 2.1 Example set of operations to be scheduled

Op #	Operation		Instruction Type	Syllable	Issue Port
1	ld	r1 ← M[r5]	M (*ii*)	M	M0, M1
2	add	r3 ← 8, r3	A	M/I	A
3	st	M[r7] ← r6	M (*iv*)	M	M2, M3
4	ld	r6 ← M[r4]	M (*ii*)	M	M0, M1
5	mov	r2 ← pr	I (*i*)	I	I0
6	add	r5 ← r1, r2	A	M/I	A
7	cmp.eq.unc	p1, p2 ← r1, r11	A	M/I	A
8	ld	r8 ← M[r6]	M (*ii*)	M	M0, M1
9	br		B	B	B0, B1, B2

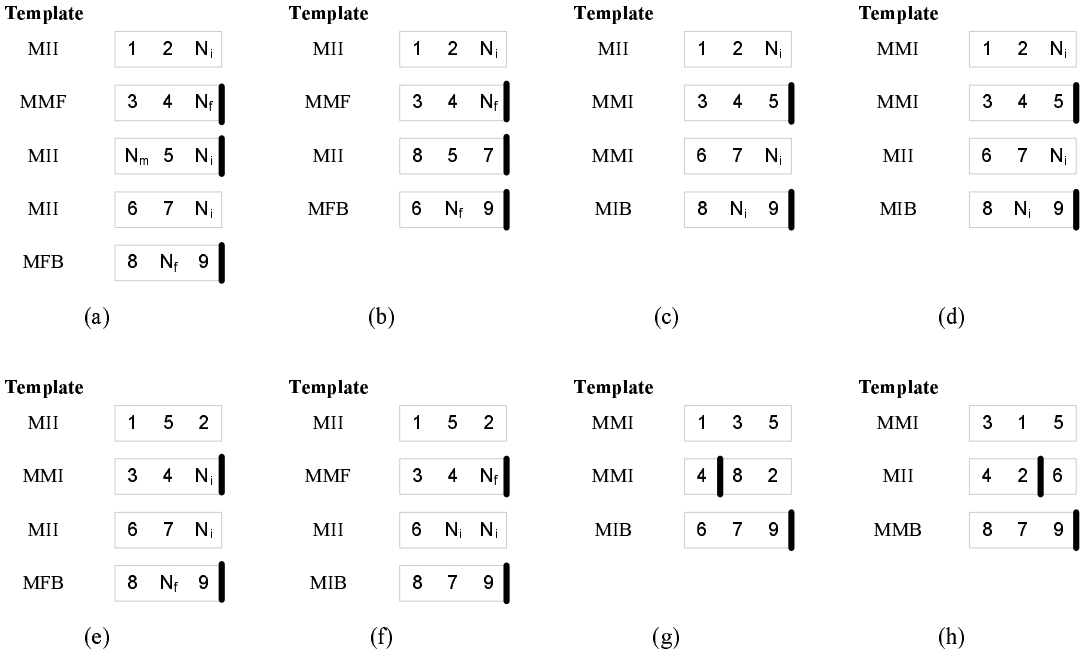


Figure 2.5 Eight legal schedule examples

the thick black bars. The operations are indicated by their operation number under the Op # column in Table 2.1 while N stands for NOP operations. The subscripts of the NOP operations denote the instruction type of the NOP operation. The instruction type of NOP operations must be compatible with the syllable of the slot, just like any normal operation. There are NOP operations for every instruction type in IPF but the A-instruction type. All of these schedules will be accepted by the assembler and the Itanium 2 hardware, and the final output will also all be the same. However, there is substantial variation amongst them. The cycle counts range from two to three cycles, and the bundle counts range from three to five. Some of the schedules use compressed templates. Sometimes the difference is as small as one syllable in the template. The cycle count will begin from cycle zero.

The performance of the schedules presented in Figure 2.5 is shown in Figure 2.6. The blue letterings above each operation is the issue port the operation is dispersed to when run on an Itanium 2. The red dashed lines indicate locations where split issue occurs, incurring an unexpected cycle boundary. The actual execution times of the various schedules range from two to four cycles, a 100% performance spread, while the number of bundles required range from three to five bundles, a 67% spread.

Judging from the dependences between the operations (keeping in mind that antidependence can be scheduled in the same cycle) and the number of slots and functional units in the Itanium 2, one would conclude that all of the instructions can be executed in two cycles. The schedules indicated in Figures 2.6(a) and (b) are scheduled for three cycles and they perform as scheduled. Figure 2.6(a) requires an extra bundle, which could affect performance if there are

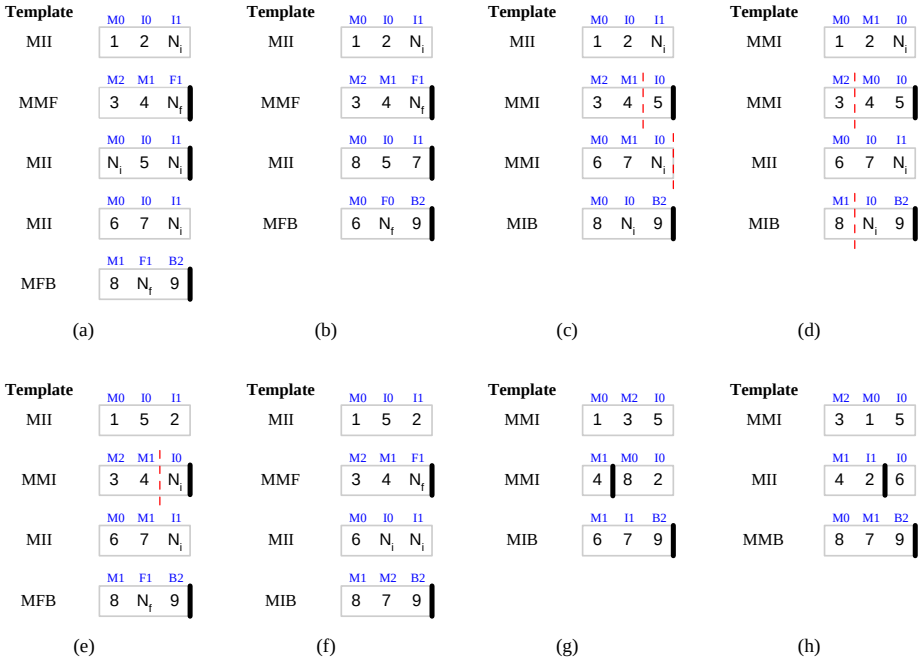


Figure 2.6 Performance of schedule examples

problems in the front end or instruction fetch. Six of the schedules, Figures 2.6(c) through (h), try to execute the operations in two cycles. Although they are all legal schedules, accepted by the assembler and hardware, they do not all execute in two cycles. Two of the two cycle schedules, Figures 2.6(c) and (d), actually take four cycles to run due to split issues. Figure 2.6(e) takes three cycles.

Instr-5 is very restrictive and can only execute on the I0-issue port. In Figure 2.6(c), Instr-2 has already taken the I0-issue port, which incurs a split issue before Instr-5. Instr-8 is a load instruction and can only execute on either the M0 or M1-issue port. However, both of those issue ports have already been taken the same cycle of Figure 2.6(c) by Instr-6 and Instr-7. This is why there is a split issue before Instr-8. Consequently, Figure 2.6(c) actually runs in four

cycles. Figures 2.6(c) and (d) schedule the operations in the exact sam locations. However, they demonstrate different split issues due to the differing template selection. In Figure 2.6(d) there is a split issue in the second bundle before Instr-4. Instr-4 is also a load instruction (M0 or M1-issue port) and both of its valid issue ports have already been taken by other operations (Instr-1 and Instr-2) before it in the cycle. The split issue after Instr-8 in Figure 2.6(d) is an example of resource oversubscription due to poor template selection. There are only two I-issue ports in Itanium 2 and they were both taken in the third bundle, therefore the NOP operation after Instr-8 causes a split issue. This situation can be resolved in two different ways: changing the template of the last bundle from MIB to MFB avoids the resource oversubscription (Figure 2.6(e)), or putting an operation that has a superset syllable in the last I-syllable slot so a M-issue port is used instead (Instr-7 in Figure 2.6(f)). This is not possible with an NOP operation because there is no NOP instruction that belongs to the A-instruction type. The split issue after Instr-4 in Figure 2.6(e) is the same resource oversubscription problem. It is resolved in Figure 2.6(f) by changing the template from MMI to MMF.

Figure 2.6(f) shows a legal two cycle schedule with no hidden mistakes that would otherwise impact the performance. Since the example contains nine operations, it may be possible to fit all of them into three bundles. Figures 2.6(g) and (h) show that this is possible using compressed templates, consequently improving code density. The main difference between them is the scheduling of Instr-2. Since there is no other operation dependent upon it, Instr-2 can be scheduled either in cycle zero or cycle one. Figure 2.6(h) is the schedule our current implementation of the template bundling algorithm would generate. It takes all of the new

constraints introduced by IPF that have been presented and discussed here under consideration when performing scheduling to avoid the potential pitfalls. Chapter 3 will present the algorithm in detail and demonstrate how it avoids the potential problems shown in this chapter.

CHAPTER 3

IMPLEMENTATION

The purpose of the template bundling algorithm is to extend the IMPACT compiler's scheduler beyond generic EPIC architectures into one that properly handles all of the new scheduling constraints introduced by IPF. Figure 1.3 shows the existing scheduler framework under the old path and the work of this thesis under the new path. The compiler can revert to the old path if it is scheduling for a generic EPIC architecture rather than one that uses a template bundling scheme like IPF. The scheduler schedules one *control block* at a time. A control block is a sequence of instructions with a single entry point and multiple exits. The current scheduling algorithms used are a variant of the Dependence Height and Speculative Yield (DHASY) algorithm [6, 7] and a modulo-scheduling algorithm [8] for acyclic and cyclic scheduling, respectively. The new path retains the modular approach of the old path, distributing the responsibilities between the scheduling algorithm, the scheduling manager, the resource manager, and the machine description. However, certain responsibilities have changed owners and the interfaces between the various parts are changed accordingly.

The machine description has been enhanced to express the new constraints introduced by IPF. The resource manager is also enhanced to ensure that dispersal rules are handled correctly. Finally, the scheduling manager contains the template bundling algorithm that oversees the slot placement of operations and formation of bundles with valid templates. The template bundling

algorithm helps the resource manager adhere to the dispersal rules by ensuring that all previous slots before the current slot have been scheduled and filled.

The interface between the scheduling algorithm and the scheduling manager and the interface between the scheduling manager and the resource manager have both changed. The scheduling algorithm is not directly altered, only its interface to the scheduling manager. The scheduling algorithm still determines the cycle for operations, but the physical placement within the cycle, or the slot placement, is now under the control of the scheduling manager. In particular, the scheduling algorithm used to specify a slot range to the scheduling manager for placement of the current instruction. The scheduling manager, in turn, used to pass the slot range to the resource manager as part of the resource query under the old path. Now, since the scheduling manager controls the slot placement, it queries only one slot at a time during the resource query. New information to model the IPF constraints such as instruction type, syllable type, and templates, are now also passed from the scheduling manager to the resource manager.

This chapter discusses the implementation of the template bundling algorithm with a bottom-up approach. The mechanisms for describing the machine-specific IPF constraints are covered first in Section 3.1. Section 3.2 describes the enhancements of the resource manager. The discussion then moves to the scheduling manager in Section 3.3, detailing the template bundling algorithm, walking through the algorithm with an example, presenting an analysis of the algorithm, and covering the compaction process. Finally, some alternate methods and implementation decisions are discussed in Section 3.4.

3.1 Machine Description Enhancements

Compilers for in-order architectures require a machine description to perform precise scheduling. A unique description is needed for each new architecture. In order to describe and implement the new scheduling restrictions from IPF in a way that can be utilized and expanded upon if future architectures adopt similar template bundling methodologies in their ISA, the new scheduling restrictions are described in the machine description. This made sense because many of the restrictions are related to hardware resources and are implementation specific.

The machine description system used by the IMPACT compiler is based upon bit vectors [2, 9]. Figure 3.1 shows graphically a portion of the machine description. The parts in blue remain from the old path. Everything in black comes from the new path to describe the IPF constraints. Every instruction in the ISA, indicated by its opcode, is described in the machine description with the resources upon which it is legal to execute, its *scheduling alternatives*. The scheduling alternatives contain not only the valid resources like slots and functional units, but also the execution latency of the instruction. The current implementation of the machine description is capable of selectively constraining the behavior of certain instructions by specifying a specific range of legal slots and functional units, like the example with branch instructions in Section 2.1. However, it cannot handle the concept of the order of operations in a cycle affecting resource usage, like that stipulated in the dispersal rules. The machine description also did not support other IPF constraints like bundles, templates, syllables, and dispersal rules.

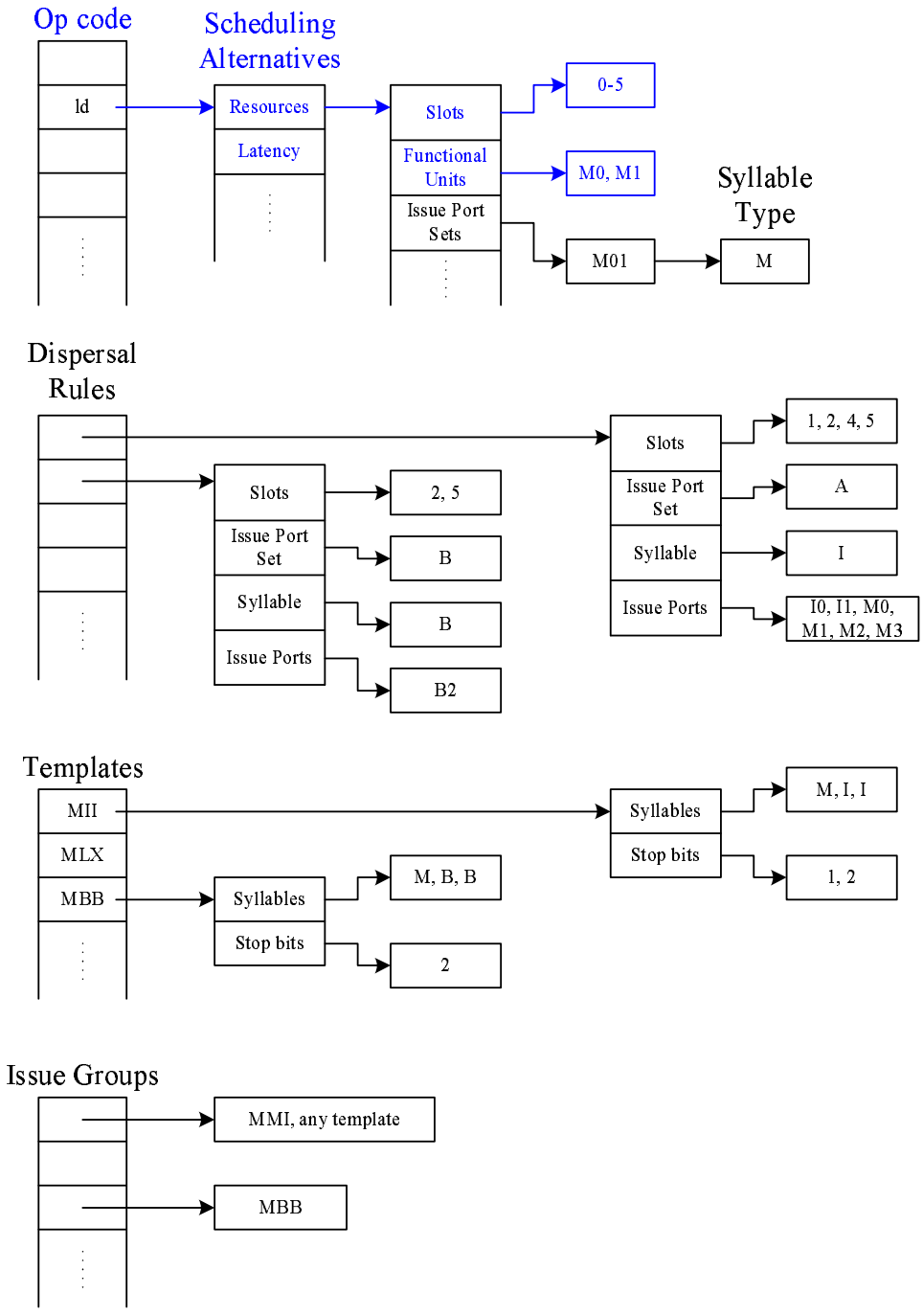


Figure 3.1 Machine description structures

Functional units are the lowest level of machine resources that the machine description is aware of. Since issue ports form the lowest level of machine modeling that concerns the compiler in IPF, functional units are used to model to issue ports in the new machine description. As shown in Figure 3.1, the functional units assigned to an instruction will be the legal issue ports for the subtype of that instruction, like the ones in Figure 2.4. However, this does not easily express the instruction's type and subtype for the purposed of checking dispersal rules. Therefore a new abstraction, the *issue port set*, is introduced. An issue port set expresses the legal issue ports of an instruction and provides a way to label the instruction with its instruction subtype and syllable type. In the current implementation for Itanium 2, issue port sets consist of the following: M2, M23, M01, M, F, I0, L, I, B, and A. The letters correspond to instruction types. If no number follows the letter, all of the issue ports associated with that instruction type are included. For example, F includes both F-issue ports, F0 and F1. If numbers follow the letter, they represent a range of issue ports. M2 corresponds to instruction subtypes that can only be dispersed to the M2-issue port, such as `alloc` on Itanium 2. This would be subtype *iii* under the M-instruction type in Figure 2.4. M23 represents the subset of M2 and M3-issue ports, like the instruction subtype for integer store instructions (subtype *iv*). M03 contains all M-issue ports: M0, M1, M2, and M3. The issue ports in an issue port set do not have to be a contiguous range of issue ports. They happen to be that way due to the various subtypes of Itanium 2. Issue port set A includes all of the issue ports associated with A-instructions, i.e., all of the M and I-issue ports. L represents the two combinations of two issue ports, F0 and I0 or F1 and I1, that execute a single L-instruction. To be more general and support future

architectures, the implementation allows an instruction to require any arbitrary number of issue ports to execute.

The order in which the issue port sets are enumerated in the machine description is used as the basis for the priority queue in the template bundling algorithm. Intuitively, the smaller the set of legal issue ports the more restrictive the scheduling for the instruction, consequently the higher its scheduling priority. The order of issue port sets presented earlier is the current priority ranking from highest to lowest. For instance, all M2, M23, and M01 instructions in the cycle should be considered before looking at M03 instructions. The role of the priority queue will become more apparent when the template bundling algorithm is presented in detail in Section 3.3.1.

As mentioned before, although IPF groups operations into bundles and expects to issue some number of bundles per cycle, it can still be viewed as an n-issue machine, n being six in the current implementations. The machine description under the old path of the IMPACT compiler, being an EPIC compiler, already has the concept of slots defined. Rather than enforce bundles as the new mechanism for operation dispersal, bundles are implemented on top of the existing slot description. All bundles are assumed to consist of the same number of slots and the slots in a cycle are distributed evenly amongst the bundles; syllables like the L-syllable type in IPF are accommodated by describing them as using two slots. The position of syllables and slots in the different bundles can be easily translated into slots in a cycle. This works quite well without major changes to the machine description set up, the only restriction being that all bundles must consist of the same number of slots.

Figure 3.1 shows how dispersal rules are modeled in the machine description. Dispersal rules are identified by the following parameters: (a) slot number, (b) syllable type, and (c) instruction subtype. Since bundles are implemented on top of the existing slot description, the slot numbers for dispersal rules are stated as if viewing Itanium 2 as a six-wide issue machine, with the numbering starting from zero. The instruction subtype is specified using the appropriate issue port set. Notice that one of the parameters listed in Section 2.3.2 for dispersal rules is missing, specifically (d) preceding operations in the cycle. Instead of enumerating all of the possible preceding operations and their effects, they are taken into account by a combination of a guarantee from the scheduling manager that all preceding slots in the cycle will have already been scheduled and keeping track of the resources they consume in the resource manager. An array is formed consisting of the bit masks of the issue ports given as resources to the dispersal rule. The order of the array must correspond to the order given within the machine description, which must match the specifications of the hardware [4]. This allows the resource manager to try the resources in the order stated in the specifications. There are two examples in Figure 3.1. For a B-instruction type instruction scheduled in slot two or slot five, the last slot of the three slot bundle in IPF, it must disperse to the B2-issue port. For A-instruction type instructions scheduled in an I-syllable slot, the range of legal issue ports is much larger, consisting of all of the I and M-issue ports. The order is very important. Itanium 2 will attempt to dispatch the instruction to an I-issue port first before trying M-issue ports. The resource manager relies on the order being maintained in the machine description so that it follows the hardware specifications correctly.

Syllable types are currently expressed using a byte in the machine description. By assigning a basic syllable type to each bit, up to eight basic syllable types can be accommodated. For example, the least significant bit can be assigned to be the M-syllable type. Superset syllables, those compatible with more than one basic syllable type, are simply the logical OR of the included basic syllable types. More bits can be used if there are more than eight basic syllable types in the architecture.

Templates in the machine description keep track of the syllables that make up the template and their order within it. The template description also contains information on legal stop bit positions. The positions of legal stop bits are listed as after a slot number in the bundle. In Figure 3.1, it shows that the MBB template can only have a stop bit after slot two of the bundle, i.e., the end of the bundle, whereas the MII template can have stop bits after either slot one or two of the bundle. The assumption is that all combinations of valid stop bit positions can be present in a single bundle. In other words, a MII bundle can have no stop bits, one stop bit between the I-syllables or at the end of the bundle, or two stop bits.

The issue group description in the machine description is responsible for delineating valid combinations of templates. This is to prevent the split issues mentioned in Section 2.4. The need for this description is primarily motivated by the set of split issues that will always occur, such as those involving BBB and MBB templates. The actual implementation in the machine description for the IMPACT compiler does not require the user to explicitly list all of the combinations. Previous work [10] enables the user to specify the combinations in a clean and

efficient manner, relying upon an optimizer to efficiently organize and manage the internal representation.

A final addition to the machine description involves producer-consumer latency modeling. The existing machine description of the IMPACT compiler assumes that an instruction will take the same amount of time to execute regardless of the consumers of its results. However, due to hardware implementation issues such as operand bypassing, this is not always the case and Itanium 2 is no exception. Consequently, new support has been added to the machine description to model special producer-consumer latencies. Every instruction is still given a base latency. However, a list of special producer-consumer pairs and the extra latencies required for those situations is kept in the machine description. If a dependence between two operations matches an item on the list, the execution time of the producer is lengthened accordingly. For example on Itanium 2, instructions using the adder, like `add` and `cmp`, typically require one cycle to generate the result. However, if the consumer of the result is a multimedia instruction, such as `pmpy2`, the result from the previous instruction will not be available until three cycles later due to additional bypass latency.

All of this defines the infrastructure in the machine description for the constraints introduced by IPF. The definitions are set up so that they can be generalized for architectures other than IPF and enhanced for future architectures. The new enhancements add on to the existing machine description framework rather than redefine them, minimizing the amount of change to the IMPACT compiler. They also provide access to information for the higher level algorithms

in the resource manager and the scheduling manager. In particular, the definition of dispersal rules provide hooks for the resource manager to identify dispersal rules and enforce them properly.

3.2 Resource Manager Enhancements

Under the old path, the resource manager in the IMPACT compiler's scheduler takes an operation and a range of slots from the scheduling manager and, after consulting the resource map, answers if resources exist to schedule the operation in a slot within the given range at the given cycle. This is shown in Figure 3.2(a). The enhancements to the resource manager primarily deal with obeying dispersal rules. The resource manager has to know about the new IPF constraints and terminology such as templates and syllables, but its job is to utilize those definitions and to ensure that the dispersal rules described in the machine description are followed. Figure 3.2(b) shows the process of the resource manager under the new path.

Since the scheduling manager now controls slot placement within a cycle, the interface between the scheduling manager and the resource manager is changed accordingly. The scheduling manager now gives the resource manager one slot to try at a time. Information concerning the IPF constraints are also passed along, such as the templates of the cycle. Note that since the scheduling manager is responsible for the template assignment of bundles and the slot position of operations, it therefore also ensures that the syllable type of the slot from the template assignment is compatible with the instruction type of the operation before handing off to the resource manager.

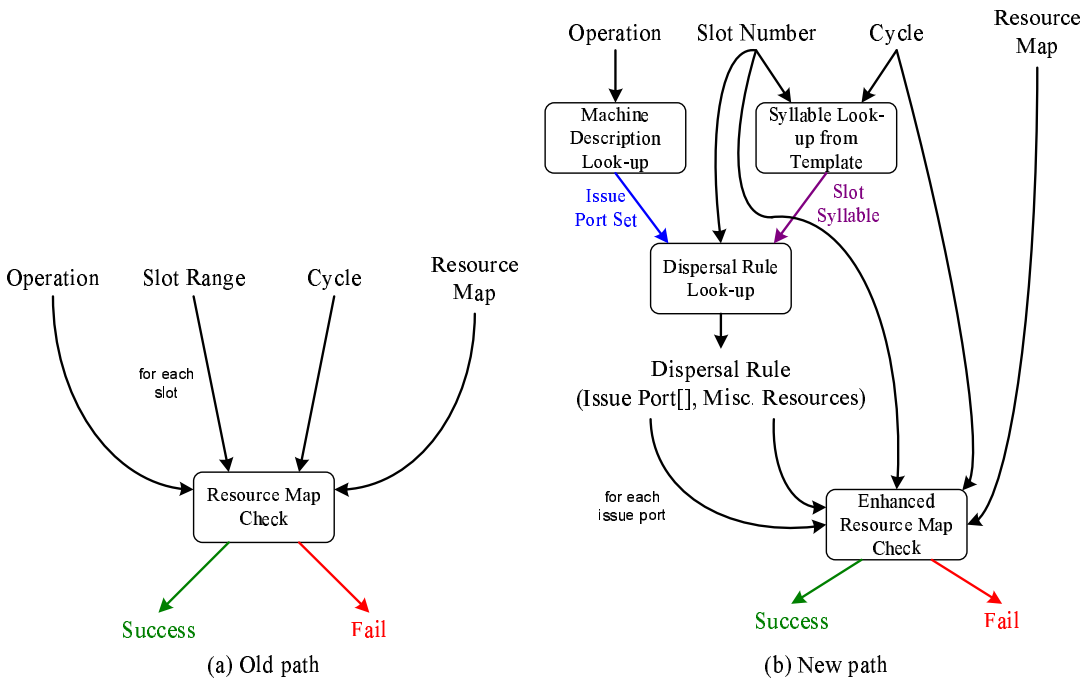


Figure 3.2 Resource manager

The resource manager looks up the subtype of the operation by looking up the corresponding issue port set in the machine description. It also gathers the syllable type of the slot using the slot number and the templates of the cycle. Using the instruction subtype (expressed as the issue port set), the slot number, and the syllable type of the slot, the resource manager finds the appropriate dispersal rule from the machine description. A dispersal rule should exist, otherwise something is wrong with the combination of slot number, instruction subtype, syllable type of the slot, and the templates of the cycle.

As discussed in Section 3.1, the dispersal rules contain an array of issue ports. The resource manager takes the issue ports from the array one at a time and checks if resources are available to schedule the operation using that issue port. As soon as an available one is found, the

resource manager stops. This emulates the left-to-right dispersal rules ordering described in the specifications. If an issue port is not available, it is taken up by an operation scheduled in a preceding slot. It is the combination of this ordered resource check along with the systematic slot scheduling in the scheduling manager that accounts for the effect of operations in preceding slots as dictated by the dispersal rules. Miscellaneous resources are other resources that must also be used. It was implemented so that the proper slot resources are marked as used in the resource map.

3.3 Scheduling Manager Enhancements

The responsibilities of the scheduling manager have increased significantly, and its interface has changed accordingly. Since the physical slot placement is now part of the scheduling manager, the scheduling algorithm now indicates only the desired cycle to the scheduling manager. Correspondingly, the scheduling manager sends only one slot to the resource manager rather than a slot range. Other additional duties of satisfying the new constraints introduced by IPF are also picked up by the scheduling manager, such as template assignments. Furthermore, the resource manager now depends on the scheduling manager for certain guarantees while scheduling to ensure that dispersal rules are met. To satisfy all of these requirements, the template bundling algorithm was devised to handle them in a systematic manner within the scheduling manager.

3.3.1 Template bundling algorithm

The template bundling algorithm is called to schedule an operation in some cycle. It considers only one cycle at a time. There may already be other operations scheduled in the cycle of interest. The algorithm will only respond with success if it is able to schedule the new operation along with all of the existing operations in that cycle. The template bundling algorithm controls the slot placement of operations within the cycle and schedules them in sequential slot order, lowest-to-highest, or left-to-right, so that the resource manager is guaranteed that the slots preceding the one it is concerned with are already occupied. This is important since the behavior of dispersal rules depends upon the operations in preceding slots.

The template bundling algorithm also determines the templates of the bundles in the cycle. It ensures that the syllable of the slot from the template is compatible with the operation being scheduled. The template bundling algorithm does not consider compressed templates as possible templates at the time of scheduling. There is a compaction phase, to be discussed in Section 3.3.2, that makes use of those particular templates. The reason for this restriction is that the scheduling manager handles one cycle at a time, as dictated by the scheduling algorithm, and compressed templates do not fit nicely within this modular model. The primary goal of the template bundling algorithm is achieving performance, as determined by the scheduling algorithm, rather than reducing frivolous NOP operations. The template bundling algorithm performs an exhaustive search throughout the operation dispersal configuration space (templates, order of operations in a cycle, etc.), so if there is a way to schedule all of the operations

in the cycle without incurring hidden penalties like split issues, it will find it. The algorithm will only return as a success if it is able to schedule all of the operations in the cycle.

The template bundling algorithm consists of four functions: (a) `Template_bundling`, (b) `Handle_backtrack`, (c) `NOP_schedule`, and (d) `Syllable_schedule`. The entry point of the algorithm is the `Template_bundling` function, which supervises the progress of the algorithm. It keeps track of the operations to be scheduled in the cycle and manages the order in which they are tried. This function also ensures that the algorithm proceeds down the slots of the cycle in a left-to-right manner. If it ascertains itself to be in a situation that leads to failure, it calls the `Handle_backtrack` function. `Handle_backtrack` takes care of the process of traversing the slots in the backwards manner, right-to-left. It constantly looks for alternatives to the schedule so the algorithm can reverse itself and resume its forward progress. The `Template_bundling` and the `Handle_backtrack` functions both rely upon the other two functions, `NOP_schedule` and `Syllable_schedule`, to perform certain duties. `NOP_schedule` is called when an NOP operation needs to be created and scheduled. There are a number of NOP operations with different attributes, specifically different syllable types, that `NOP_schedule` goes through systematically so that all possibilities are explored. `Syllable_schedule` is called when an operation, be it real or NOP, is to be scheduled in a slot. It calculates the template of the proposed permutation of operations and verifies if any valid template corresponds. It then performs the resource query by passing the information to the resource manager. The rest of this section describes the template bundling

algorithm in detail, starting by walking through the algorithm, continuing with an example and an analysis of the algorithm, and concluding with techniques for efficient implementation.

3.3.1.1 Template bundling algorithm description

The scheduling algorithm schedules one control block of the program at a time. It decides that operation `x` should be scheduled at cycle `t` in a control block. This is the information the scheduling manager, in particular the template bundling algorithm, starts with. An issue group, `g`, is associated with each cycle `t`. There may be other operations previously scheduled in `g`. They form the set `Y`. The template bundling algorithm determines if the set of operations `Y + x` can be scheduled in cycle `t` in a range of slots, `s1` through `s2`. The algorithm schedules operations into the slots of the cycle one at a time in order, starting with `s1`. This fulfills the scheduling manager's obligation to the resource manager and helps it properly observe the dispersal rules. The algorithm examines the operations in the priority queue one at a time, starting with the operation with the highest priority. It fills a slot with an NOP operation if no real operation can be scheduled in it. The algorithm calculates and adapts the templates of the bundles as it schedules operations into the slots. As the algorithm traverses down the slots, it may generate a situation that makes it impossible to complete the scheduling of all of the operations. At this point, the algorithm *backtracks* and tries other permutations of the operations. If the algorithm backtracks out of the valid slot range, lower than slot `s1`, then no valid schedule can be found. All restrictions, both IPF-specific and those general to scheduling, such as template validity, resource usage, and dependences, must be obeyed in the final schedule. If

the algorithm is unable to do so, then it will say it failed and restore the cycle, or issue group `g`. In most situations, `s1` and `s2` should be the minimum and maximum slots of the architecture in a cycle, zero and five for Itanium 2. This gives the scheduling manager the full range of slots and resources to schedule with. However, the range restriction feature was retained for code reuse purposes. The compaction phase, to be detailed in Section 3.3.2, needs to restrict the range of slots available in a cycle when scheduling with compressed templates.

Although the template bundling algorithm is generic and machine nonspecific, there are some assumptions made; the assumptions follow IPF specifications since that is the implementation's primary target. The algorithm assumes that dispersal rules follow a left-to-right pattern, meaning preceding operations in the same cycle affect the dispersal of those to come. Intracycle dependences are allowed assuming the compiler provides for correctness; i.e., the operations are scheduled in the cycle such that the order of the dependence is satisfied. Examples include antidependences and zero-cycle dependences. Finally, NOP operations are assumed to also issue and consume issue ports, thereby affecting the dispersal of useful operations. Consequently, they have to be handled with care in the template bundling algorithm. The rest of this section is a detailed description of the template bundling algorithm.

The template bundling algorithm begins with the `Template_bundling` function, as shown in Figure 3.3. It starts by querying for an issue group `g` at cycle `t` in the control block, creating one if none exists for that cycle. If `g` already exists, then there could be other operations in `g`, which form the set `Y`. It is possible for `g` to exist but `Y` to be empty because issue

```
1: Template_bundling (op x, cycle t, earliest slot s1, latest slot s2)
2: issue group g = Issue_group[t]              //Once unique issue group per cycle per control block
3: Set Y = Instruction_set (g)                 //Existing operations in g
4: if Insufficient_resources (Y+x) then
5:    return FALSE
6: end if
7: issue group g_backup = g
8: unschedule Y
9: priority queue q = Enqueue (Y+x)            //Not enough slots for remaining operations or split issue
10: i = s1
11: while s1 <= i <= s2 + 1 do
12:    if |q| > s2 + 1 − i or Split_issue (g) then
13:       i = i − 1; CONTINUE
14:    end if
15:    if for some j s.t. s1 <= j < i, op w = g[j] and j + w′ > i (w′ = number of slots required by w) then
16:       i = j; CONTINUE
17:    end if
18:    if g[i] contains an operation then
19:       op v = Handle_backtrack (x, v, t, g, i, q)
20:       if v = NULL then
21:          CONTINUE
22:       end if
23:    else
24:       if |q| > 0 then
25:          take the first operation in q as v
26:       else
27:          if i > s2 then
28:             BREAK                          //Schedule succeeded
29:          end if
30:          if NOP_schedule (g, t, i) then
31:             i = i + 1; CONTINUE
32:          else
33:             i = i − 1; CONTINUE
34:          end if
35:       end if
36:    end if
37:    while v ! = NULL do
38:       if v satisfies all restrictions and Syllable_schedule(v, g, t, i) (if v is a superset syllable call Syllable_schedule multiple
          times with v set to different compatible syllables until one works or all fail) then
39:          i = i + v′; remove v from q; BREAK
40:       end if
41:       v equals the next operation in q; v = NULL if the end of q is reached
42:    end while
43:    if v = NULL (failed to schedule any operation) and NOP_schedule(g, t, i) then
44:       i = i + 1; CONTINUE
45:    else
46:       i = i − 1; CONTINUE
47:    end if
48: end while
49: if i < s1 then
50:    g = g_backup; return FALSE              //Schedule failed
51: end if
52: return TRUE
```

Figure 3.3 Template_bundling algorithm

groups are never deleted while operations can be unscheduled in the IMPACT compiler. Empty issue groups are simply skipped over when the schedule is committed.

The resource check on line four of Figure 3.3 is a high-level resource check utilizing the subtype, or issue port sets, of the operations in `Y + x`. For example on Itanium 2, there is only one I0-issue port. Consequently, there cannot be more than one operation in the cycle with the I0-issue port set. One M2 and two M23-issue port set operations also would not fit in the same cycle either. Issue port sets therefore provide a quick resource check at the beginning of scheduling a cycle.

If the resource check passes, the current configuration of `g` is saved in `g_backup` in case the algorithm fails for other reasons. All of the operations in `Y` are unscheduled and all `Y + x` operations are put into a priority queue `q`. The priority of an operation is based on its subtype, the issue port set. The order of the priority was shown in Section 3.1 when issue port sets were introduced. The priority of an operation will never change, since it is a property of its instruction subtype. Therefore, if an operation is taken out of the priority queue and inserted back, it will be close to its original position. Its position can differ only with respect to other operations with the exact same priority.

The algorithm initializes the index variable `i` to the lowest slot allowed `s1`. As long as `i` remains in the range of valid slots the algorithm continues. The algorithm does not stop automatically when it runs out of real operations for the cycle. It fills the rest of the slots, up to and including `s2`, with NOP operations to ensure that there are no hidden split issue concerns. If the algorithm backtracks out of the valid slot range, lower than slot `s1`, however,

the algorithm stops because that is the indicator that no valid schedule exists for the operations `Y + x`.

Line 12 in Figure 3.3 perform some checks that tell the algorithm to backtrack. If the number of operations left in the priority queue to be scheduled $|q|$ is greater than the number of slots left, $s2 + 1 - i$, then the remaining operations can not possibly fit in the issue group. The algorithm also utilizes the issue group specification in the machine description (Section 3.1) to check for split issues due to template combinations. If the combination of templates in `g` is not listed as valid in the machine description, `Template_bundling` begins to backtrack.

Since operations may take up more than one slot (e.g., L-instructions in IPF), it is possible for the algorithm to place the index `i` in a slot that appears empty but is actually occupied by one of these multislot instructions. This only occurs when backtracking because the algorithm ensures that the index `i` is incremented correctly when scheduling forward. Therefore, as seen on line 15, the algorithm searches for an operation scheduled in a prior slot to make sure `i` is not sitting in someone else's slot. The notation `g[j]` means the content of slot `j` in issue group `g`. If `i` is in a slot that belongs to an operation `w` in slot `j`, the algorithm decrements `i` to `j` and continues to backtrack.

The algorithm increments `i` when it successfully schedules an operation or NOP operation in a slot. The index `i` is decremented only when the algorithm is backtracking. Since the algorithm starts from the lowest available slot, if an operation is present in slot `i` of `g`, `g[i]`, then the algorithm must be backtracking. A decision must be made if something can

```
1: Handle_backtrack (op x, time t, issue group g, int i, priority queue q)
2: op z = g[i]                          //Get the operations in slot i of g
3: if z is a NOP operation then
4:    if NOP_schedule (g, t, i) then
5:       i = i + 1; return NULL        //Replaced z with new NOP with different syllable
6:    else
7:       Unschedule and delete z; i = i − 1; return NULL
8:    end if
9: else
10:   if z is a superset syllable then
11:      if Syllable_schedule (z, g, t, i) succeeds with another syllable type then
12:         i = i + z′; return NULL
13:      end if
14:   end if
15:   unschedule z and enter into q by its priority
16:   v = operation after z in q; v = NULL if z is the last operation in q
17:   if v = NULL then
18:      if NOP_schedule (g, t, i) then
19:         i = i + 1
20:      else
21:         i = i − 1
22:      end if
23:      return NULL                   //No more operatoins, tried to insert NOP operation
24:   end if
25:   return v
26: end if
```

Figure 3.4 Handle_backtrack algorithm

be done in the current slot to proceed forward, or to continue backtracking. This is done by the `Handle_backtrack` function in Figure 3.4 (called on line 19 in Figure 3.3). `Handle_backtrack` either takes care of the situation and adjust `i` appropriately, or it returns a new operation, `v`, for `Template_bundling` to try scheduling in the current slot.

`Handle_backtrack` (Figure 3.4) first obtains the operation `z` in `g[i]`. Since the template algorithm tries to schedule an operation from the priority queue `q` before filling a slot with a NOP operation, it means no operation in `q` can be scheduled in `i` if `z` is already a NOP operation. However, it may be that a NOP operation with a different syllable type can be scheduled in `i`. This is handled by the `NOP_schedule` function (Figure 3.5). If it returns with a new NOP operation with a different syllable type, `i` is incremented to stop the backtracking.

```
1:  NOP_schedule (issue group g, time t, slot i)
2:  op o = g[i]
3:  if o != NULL then
4:     syllable type y = Get_syllable_from_op (o)   //Get the syllable type
5:     int j = Syllable_to_NOP_index (y)            //Get the NOP_index
6:     j = j + 1                                    //Increment the NOP_index since it failed
7:  else
8:     j = 0
9:  end if
10: template p = Get_template (g, i)
11: while j <= j_MAX (the last NOP instruction type available for the architecture) do
12:    if p is locked then
13:       if o != NULL then
14:          return FALSE
15:       end if
16:       syllable s = Get_syllable_from_template (p, i)
17:       op n = Create_NOP (s)
18:       j = j_MAX                                 //Only one syllable allowed if template is locked
19:    else
20:       syllable s = NOP_index_to_syllable (j)
21:       template p2 = Calculate_template (p, s, i)
22:       if Valid_template (p2) then
23:          op n = Create_NOP (s)    //Create NOP based on NOP_index
24:       else
25:          j = j + 1; CONTINUE
26:       end if
27:    end if
28:    if Syllable_schedule(n, g, t, i) then
29:       g[i] = n
30:       if p is not locked then
31:          New_template (g, i, p2)
32:       end if
33:       return TRUE
34:    end if
35:    j = j + 1
36: end while
37: return FALSE
```

Figure 3.5 NOP_schedule algorithm

If `NOP_schedule` fails, `z` is deleted, `i` is decremented, and the algorithm will continue to backtrack.

If `z` is not an NOP operation, it must have come from `q`. If `z` has a superset syllable, it is possible that `z` can be scheduled in the current slot `i` under a different compatible syllable type with a different template. The `Syllable_schedule` function (Figure 3.6) is called to verify this. If successful, `i` is incremented by the number of slots required by `z`, z', and the template bundling algorithm ceases to backtrack. If `z` does not have a superset syllable or no other compatible syllable succeeds, it is unscheduled and put back into `q`. Recall that `z` will go back into `q` at approximately the same position it occupied before since the queue is ordered by the priority of the operations. If there are no operations in `q` after `z` (i.e., `v` is not valid), that means all of the operations in `q` have been examined and none are appropriate for slot `i` given the current situation. Consequently the algorithm tries to schedule a NOP operation in slot `i`. If that succeeds, the algorithm proceeds. If it fails, the algorithm continues to backtrack. If `v` is valid, it is returned out of `Handle_backtrack` as the next operation to try. For all other situations, NULL is returned.

Certain operations have a superset syllable as their syllable type. This means they are compatible with more than one syllable type and can be scheduled in all slots with the appropriate syllables. The template bundling algorithm leaves the determination of templates until fairly late, either `NOP_schedule` or `Syllable_schedule`. In fact, the templates are determined by the operations as they are scheduled. However, this also means that the algorithm has to

```
 1: Syllable_schedule (op x, issue group g, time t, slot i)
 2: template p = Get_template (g, i)
 3: if p is locked then
 4:     syllable type s = Get_syllable_from_template (p, i)
 5:     if s is not compatible with op x then
 6:         return FALSE
 7:     end if
 8: else
 9:     syllable type s = Get_syllable_from_op (x)
10:     template p2 = Calculate_template (p, s, i)
11:     if !Valid_template (p2) then
12:         return FALSE
13:     end if
14: end if
15: if Query_resource_manager (x, g, t, i) then
16:     g[i] = x
17:     if p is not locked then
18:         New_template (g, i, p2)
19:     end if
20:     return TRUE
21: end if
22: return FALSE
```

Figure 3.6 Syllable_schedule algorithm

have some way to systematically test operations with a superset syllable, treating it as having a different syllable type each time. The internal representation of the machine description helps the scheduling manager do this. Each bit of the syllable type representation is associated with a specific syllable type. Therefore, superset syllables are the only ones with more than one bit set at a time. Whenever an operation with a superset syllable is encountered, the least significant bit of its internal syllable type representation is extracted, and it is scheduled as if it is of that syllable type. If the scheduling fails or the operation is under consideration again while backtracking, the next least significant bit of its internal representation is extracted and used. Thus for each slot, the operation is considered once for each syllable type with which it is compatible. For example in IPF, A-instruction type operations have a superset syllable that

is compatible with both M- and I-syllable types. Assume that the least significant bit represents the M-syllable type and the next bit I-syllable type. When an A-instruction type operation is first being scheduled, it is treated as having an M-syllable, meaning the syllable of the slot from the template must be M. If this fails, either due to resource constraints, invalid template types, backtracking, etc., the operation is considered again with the next bit, i.e., as an I-syllable. If the operation is under consideration in the same slot for the third time, there would be no more bits left in the internal representation and the algorithm will know it has performed the exhaustive search on the operation.

Coming back to line 24 in `Template_bundling`, `g[i]` does not already contain an operation. If there are operations left to be scheduled, $|\texttt{q}| > \texttt{0}$, the first operation, the one with the highest priority, is taken out of `q` and used as `v`. If there are no operations left to be scheduled, the algorithm continues by scheduling NOP operations in the remaining slots. This is because NOP operations are assumed to issue and consume resources, and it is the algorithm's duty to ensure that no split issue occurs. Since this potentially creates bundles with only NOP operations, empty bundles are culled when the schedule is finalized.

If `v` is valid by line 37, this means there are operations left to be scheduled. The algorithm attempts to schedule the operation by calling `Syllable_schedule`, moving on to the next operation in the priority queue `q` if it is not successful with the current operation. If an operation out of `q` has been scheduled in the current slot `i`, the algorithm continues. Otherwise, it attempts to insert a NOP operation instead after attempting and failing to schedule the operations in `q`. If the algorithm cannot insert an NOP operation either, then the current situation is

a dead end, and the algorithm begins to backtrack. Finally, when the algorithm exits the main while loop in `Template_bundling` spanning lines 11 through 48 in Figure 3.3, it either restores the issue group from `g_backup` if it failed or return success.

The `NOP_schedule` function tries each type of NOP operation once for each slot. It does this by using the *NOP index*. All of the NOP operations are identified in the machine description and given an index number. By being able to obtain the index number from the syllable of the slot (line five in Figure 3.5) and trying each index number only once, `NOP_schedule` is guaranteed to try all of the NOP operations available to the architecture without repeating itself. Special attention needs to be paid if the template of the issue group is locked. This only happens during the compaction phase (Section 3.3.2). If the template is locked, there can only be one syllable type for that slot. Therefore, `NOP_schedule` tries only one syllable type. At the end, the template is updated with the new NOP operation unless the template is locked.

3.3.1.2 Template bundling algorithm example walk-through

This section examines how the algorithm behaves when applied to the example code sequence presented in Table 2.1. For simplicity, it is assumed that the compiler has determined that none of the memory instructions will ever alias with each other and therefore may be freely rearranged with respect to each other. In reality, pointer analysis must provide the information regarding the dependences between memory instructions. The IMPACT compiler also assumes that all memory accesses will hit in the cache; therefore, it schedules for the minimum latency. The target architecture will be the Itanium 2.

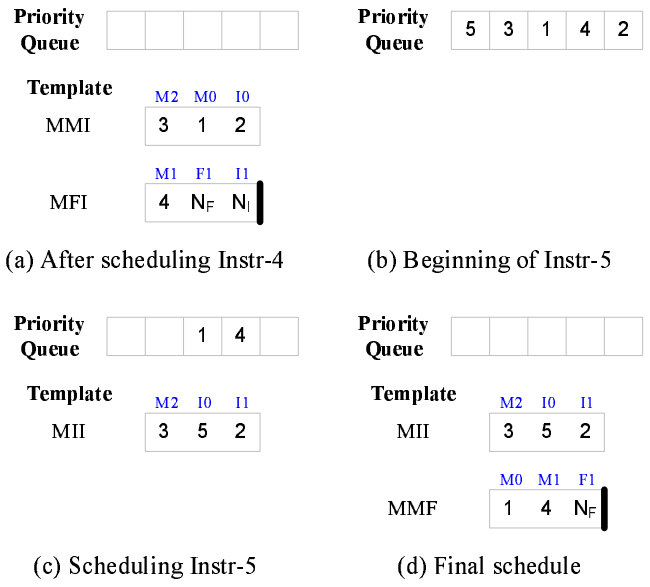


Figure 3.7 Scheduling cycle 0

Judging from the dependences among the instructions, the scheduling algorithm would determine that Instr-1, 2, 3, 4, and 5 can all be issued in the first cycle, cycle zero. Let us assume that this is also the order they will be tried in. (An actual scheduling algorithm may assign different scheduling priorities to the instructions.) After scheduling Instr-4, the schedule will look like Figure 3.7(a). The reason Instr-3 is scheduled in slot zero is because the M2 and M3-issue port subset is heuristically determined to be more restrictive than the other subsets; consequently, it is placed towards the beginning of the priority queue in the template bundling algorithm.

Instr-5 requires special attention since it has the very restrictive issue port set of I0. The earliest slot `s1` is 0 and the latest slot `s2` is 5 for the Itanium 2. The high level resource check passes and all of the currently scheduled instructions in `Y`, Instr-1 through 4, are unscheduled.

All instructions, `Y + x`, are put into the priority queue `q`, with the order from highest priority to lowest being: Instr-5, 3, 1, 4, then 2. This is shown in Figure 3.7(b). The algorithm first attempts to schedule Instr-5 into slot zero, but fails on line 11 in `Syllable_schedule` (Figure 3.6) since there is no template in IPF that begins with the I-syllable. The algorithm moves on to the next instruction in the queue, Instr-3, and succeeds in scheduling it into slot zero. The index `i` is incremented by one and Instr-3 is removed from `q`.

The current situation passes the checks on lines 12, 15, and 18 in Figure 3.3 so Instr-5, the first operation in `q` is again taken as `v`. Instr-5 is scheduled into slot one, since templates exist with the M-syllable in slot zero and the I-syllable in slot one, such as MII. Since no template in IPF ends in the M-syllable, neither Instr-1 nor 4 can be scheduled into slot two. However, Instr-2 can be scheduled in slot two since it has the superset syllable A. At this time the remaining operations in the priority queue are Instr-1 and 4 (Figure 3.7(c)). The final schedule for cycle zero is shown in Figure 3.7(d).

Scheduling for cycle one is fairly straightforward. Figure 3.8(a) shows the priority queue when scheduling Instr-9. Instr-8 ends up before 6 and 7 because the M01-issue port set has a higher priority than A, since A-syllable instructions are very versatile and can be dispatched to any M- or I-issue port. The final schedule for cycle one is shown in Figure 3.8(b). Instr-9 follows all the other instructions because no operation should cross the branch. The final schedule for the example operations is shown in Figure 3.9.

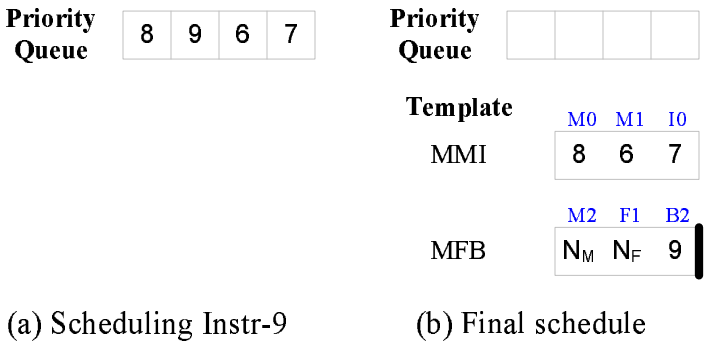


Figure 3.8 Scheduling cycle 1

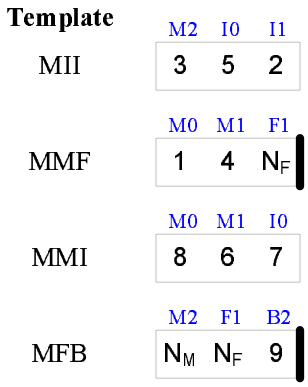


Figure 3.9 After template bundling algorithm; before compaction

3.3.1.3 Template bundling algorithm analysis

The template bundling algorithm involves the following search spaces:

- Syllable type. An architecture has s number of basic syllable types; five in the case of IPF. Superset syllables are compatible with some number of syllable types in s. The total number of syllable types, basic plus superset, will be termed S; six in the case of IPF.

- Template type. A template is a permutation of a number of basic syllables. The maximum number of syllables N that form a bundle is architecture specific; three for IPF. Each architecture specifies a number of valid template types T; IPF currently has 10 basic template types. Compressed templates are not part of this search space since the template bundling algorithm delays their usage until the compaction phase. If compressed templates are to be considered, then the search space is increased by the number of compressed templates available. However, the number of syllables available per template N will be smaller for compressed templates, unless the single cycle approach is eliminated. The compaction phase will be discussed in Section 3.3.2.

- Issue groups. The maximum number of bundles that can be issued in one cycle, or issue group, is represented by B and is architecture specific. Current implementations of IPF contain up to two bundles per issue group. Certain combinations of valid bundles can not be executed in the same cycle, causing split issue (Section 2.4). These restrictions are defined in the machine description and is a low constant referred to as I.

- Dispersal rules. The dispersal rules are modeled as additional constraints in the resource map and incur a small increase in that search space. The computational complexity of the resource map is specific to the compiler and will be referred to as R. Note that accessing the resource map to schedule an operation, incurring the computation time of R, is necessary, even if template bundling is not required for the architecture.

- Valid NOP types. The number of valid NOP types P will be equal to or less than the number of syllable types s. IPF has five NOP types, corresponding to the five basic syllable types. In other words, the superset syllable A does not have a corresponding NOP instruction. NOP operations are assumed to issue according to the dispersal rules and consume resources, just as if they are a valid operation of the same syllable. This is a trait of the IPF architecture and is an additional constraint.

If the architecture allows N syllables per bundle and up to B bundles per issue group, that means a total of $N \times B$ operations can issue each cycle. This will be referred to as as U since it will be used quite often. For the upper bound analysis, all U operations have a superset syllable type that is compatible with all s basic syllable types and all permutations of all syllable types correspond to a valid template in T. Effectively, the number of scheduling possibilities is a $\mathbf{k-permutation}$, ${}_nP_k = \frac{n!}{(n-k)!}$, where $n = sU$ and $k = U$ ($n = sU$ since the algorithm tries all compatible syllable types for each operation). Although this does not explicitly consider NOP operations, the search space involving NOP operations is less since $P <= s$.

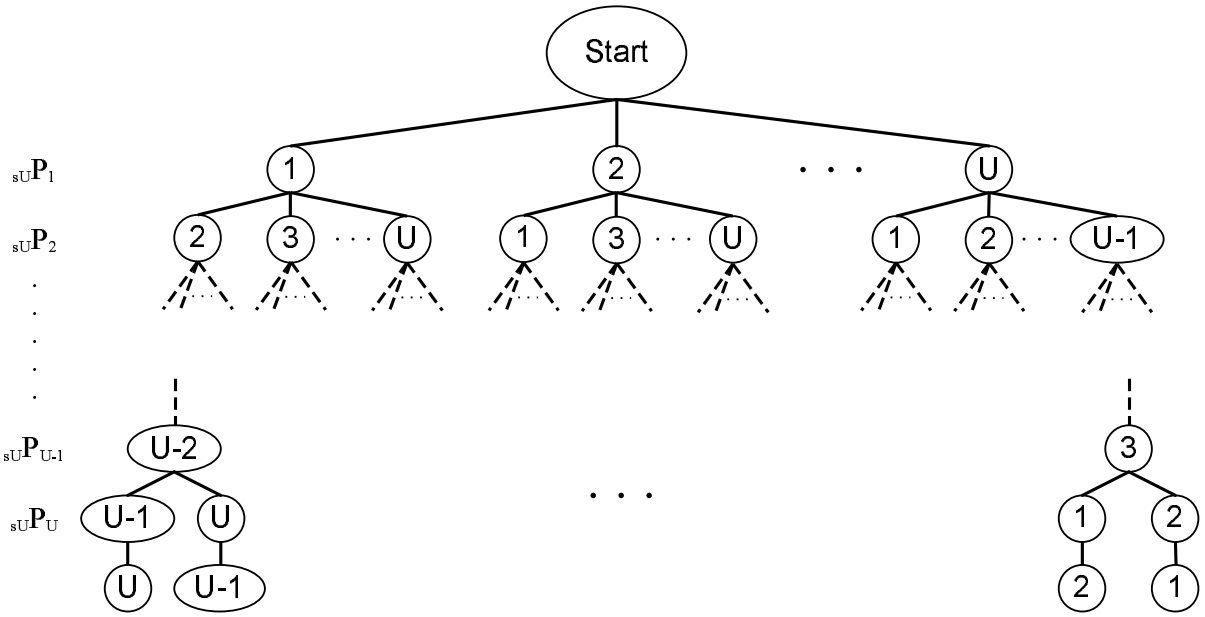


Figure 3.10 Search space of the template bundling algorithm

Figure 3.10 shows the search space of the template bundling algorithm. The algorithm begins at "Start," and traverses down the search space by scheduling one operation at a time. Note that on the first level of the search space, there are ${}_{sU}P_1 = (sU)!$ possibilities, not just U. This is because the upper bound analysis assumes that all U operations have a superset syllable compatible with all s syllable types. All other levels are affected similarly. When an operation is scheduled, it checks for split issues and calls `Syllable_schedule`, incurring the computation time of ITR. Traversing the entire search space involves the following computation complexity:

$$ITR({}_{sU}P_U + \sum_{k=1}^{U} {}_{sU}P_k)$$

In practice, the situation is much better. In IPF, there is only one superset syllable A which is compatible with only two basic syllable types. Consequently, instead of $n = sU$, for IPF $U <= n <= 2U$. Template definitions would be practically meaningless if all permutations of all syllables are valid, which means many permutations are discarded quickly. Many permutations are also discarded due to stringent resource requirements, like operations with restrictive issue port sets like M2. The algorithm traverses the search space such that sections of it are truncated as soon as a problem is detected. When the algorithm fails to schedule an operation (v on line 38 of Figure 3.3) and moves to the next operation in the priority queue, it proceeds horizontally in the search space at the same depth. The search space that extends downwards from the previous operation is effectively culled. If the algorithm is unable to schedule an operation from the priority queue and backtracks, it is moving upwards in the search space and will never revisit that particular branch again. I is a low constant for IPF, and R is present even if template bundling is not required of the architecture. Again, the addition of dispersal rules to R only increases its complexity marginally compared to what is already there. Suggestions for improving the template bundling algorithm to reduce compilation time will be covered in Section 6.1. However, there are also implementation techniques that have been utilized that greatly reduce the compilation time.

3.3.1.4 Efficiency in implementation

This section goes over implementation techniques for the template bundling algorithm that reduced the compilation time. Rather than reduce the complexity of the algorithm, which is

reserved for future work, these optimizations deal with compilation concerns such as keeping information within the compiler up to date in an efficient manner. These optimizations had large effects on the compilation time and are arguably more important than reducing the complexity of the template bundling algorithm since the search space is restricted in many areas by architecture specifications.

In IPF, certain intracycle dependences are permitted. Specifically antidependences and certain flow dependences are allowed to exist within a single cycle. The scheduling algorithm is aware of the dependences between operations and gives an operation to the scheduling manager only if its incoming dependences are resolvable by the cycle given. In other words, all of the operations in the set `Y + x` (Figure 3.3) should have all of their incoming dependences resolved except for intracycle dependences. Therefore, one implementation optimization of the template bundling algorithm is to begin backtracking as soon as any dependence is violated. Since it is established that all of the dependences are resolvable, the only dependences that can be violated are intracycle dependences. By backtracking as soon as a dependence is found to be violated, the number of permutations of operations that need to be checked is effectively reduced. In the current implementation, the dependences of an operation are checked in line 38 in Figure 3.3. If scheduling `v` in the current slot `i` violates some dependence, that means some other operation has been scheduled earlier in the cycle that must come after `v` instead. Therefore, instead of trying other operations remaining in the priority queue, the algorithm exits the while loop on line 37 in backtrack mode, decrements `i`, and continues on in the main while loop (line 11).

Another implementation optimization based on dependences was found to significantly decrease compile time. When an operation is scheduled or unscheduled, all of its dependences must be updated. This could be considerable if the operation has many dependences. Some examples include branch operations with flow dependences to all other operations in the control block and memory operations with alias dependences with other memory operations. Currently, operations are scheduled and unscheduled and its dependences updated in the main while loop of the `Template_bundling` function (Figure 3.3) every time `Syllable_schedule` is called. If an operation has many dependences and is repeatedly scheduled and unscheduled again and again as the algorithm goes through the various permutations of operations in the slots, a lot of time would be wasted on useless work. However, since it is established that the operations in `Y + x` should have their dependences resolvable other than intracycle dependences, the algorithm can restrict the dependences that are updated to only those between operations within `Y + x` instead. Consequently, the number of dependences that are updated decreases dramatically, and dependences to operations not in `Y + x` are updated all at once at the end of the `Template_bundling` function.

3.3.2 Compaction

The purpose of the compaction phase is to utilize the compressed templates to reduce the code size. It assumes that the starting schedule is as compact as it can be without using compressed templates. The two phase approach was adopted so that the template bundling

```
1: Compaction_phase (control block cb)
2: s_MAX = Architecture_max_slot ()
3: for each issue group i in cb do
4:    if i is the last issue group then
5:       BREAK;
6:    end if
7:    t = Get_cycle (i)
8:    j = Next_non-empty_issue_group (cb)
9:    if i is not as compact as can be due to arbitrary unscheduling then
10:      Unschedule one operation x from i
11:      Template_bundling(x, t, 0, s_MAX)
12:   end if
13:   if Not_good_for_compressed_templates (i, j) then
14:      CONTINUE;
15:   end if
16:   for each compressed template type c do
17:      i_backup = i
18:      Lock_template_of_last_bundle (i, c)
19:      Unschedule one operation x from i and get slot s2 right before the stop bit in c
20:      if Template_bundling(x, t, 0, s2) then
21:         j_backup = j
22:         Lock_template_of_first_bundle (j, c)
23:         Unschedule one operation x from i and get slot s1 right after the stop bit in c
24:         if Template_bundling(x, t, s1, s_MAX) then
25:            BREAK;
26:         end if
27:      end if
28:      i = i_backup
29:      j = j_backup
30:   end for
31: end for
```

Figure 3.11 Compaction algorithm

algorithm can be more general. The compaction algorithm retains the original scheduled performance since it does not change the intended cycle of operations. The difference in NOP operations with and without the compaction phase is presented in Chapter 4.

In IMPACT, the scheduler handles one control block at a time. After each control block is scheduled it goes through the compaction phase to utilize compressed templates to reduce code size. Figure 3.11 shows the `Compaction_phase` algorithm. The compaction algorithm steps through the control block one issue group at a time. It finds the next issue group in time and sees if there is a possibility that the two issue groups can share a compressed template.

The checks performed on line 13 of Figure 3.11 are architecture specific checks dependent upon the compressed templates available. Since the set of compressed templates is probably a small portion of the set of valid templates, certain characteristics may be derived to narrow down the legal cases that would beneficial. For example, the only compressed templates currently available on IPF are based off of the MII and MMI templates. One easy check is the number of operations in `i` and `j`. For IPF, if the number of operations currently scheduled in the two issue groups of interest total more than nine, then there is no compressed template that will reduce the number of NOP operations. Another situation is if either `i` or `j` contains three or fewer operations with syllable types like B, F, or L, syllable types that are incompatible with the existing compressed templates.

If the checks pass, the compaction algorithm changes the appropriate template of the current issue group `i` to a compressed template, locks it, and reschedules the operations within the entire issue group using the `Template_bundling` function (Figure 3.3). If it succeeds, the compaction algorithm attempts the same thing with the following issue group, `j`. If either reschedule fails, the algorithm tries the other compressed templates. Note how the slot ranges specified for the `Template_bundling` function calls are not simply zero and the maximum slot anymore. This is to force operations to remain on the correct side of the stop bit in the compressed template. Figure 3.12 shows the final schedule for the example operations after compaction. Notice how all of the operations remain in the cycle they were originally scheduled for in Figure 3.9. However, the three NOP operations were removed and the example now consists of three bundles instead of four.

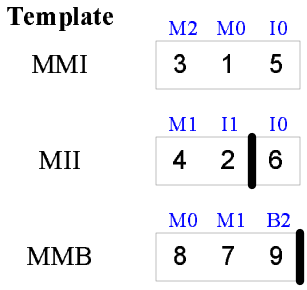


Figure 3.12 Schedule after compaction

3.4 Implementation Alternatives

Several decisions were made during the implementation of the template bundling algorithm that had significant impact. This section discusses some implementation alternatives that could have been done instead. These range from organizing the scheduler structure differently to tweaking and taking advantage of certain aspects of the IPF constraints.

Rather than keeping the modular approach, the new scheduler could have been an integrated unit instead. In particular, integrating the scheduling algorithm and molding it for the IPF restrictions would mean great changes to the implementation and possibly the results. However, creating an IPF-specific compiler was not the intention of this work. The idea was to add extensions to the existing IMPACT compiler so that it can be configured to either compile for generic EPIC architectures or IPF-like template architectures. Furthermore, the modular approach allows easier adaptation to changes for future architectures and experimentations with different scheduling algorithms. Integrating the scheduling algorithm would have made these other objectives much harder to achieve. There has been other proposal to do this integration, which will be covered in Chapter 5.

Currently, the motivation for the two phase approach lies in delaying the integration of compressed templates. Other works [3] bypass this by considering compressed templates at all times in the scheduler. The consequences of this decision are not very clear. As presented in Section 3.3.2, the complexity of the compaction phase algorithm is relatively low, being a linear pass through the control block. However, the scheduler can arguably take better advantage of scheduling slack to generate better code if it knows about compressed templates at all times. The decision was made to proceed with the two phase approach for primarily two reasons: (a) ease of implementation and separation of concerns and (b) the IMPACT compiler's assumption of the scheduling manager handling one cycle at a time.

Theoretically, by exploiting split issue judiciously, one can achieve better code density at the expense of code legitimacy. However, a conscious decision was made not to do so. Although a description of the hardware behavior when split issue and resource oversubscription occurs is described in the Itanium 2 specification, it is by no means a guarantee. It is, after all, improper behavior and can be considered a failure on the part of the compiler. Furthermore, future architectures may not guarantee correct behavior in the case of split issues.

The existing IMPACT compiler infrastructure, the modular scheduler and the bit-vector based machine description for example, had great influence on the implementation of the template bundling algorithm. The descriptions for the new IPF constraints were integrated into the existing machine description, and the template bundling algorithm was implemented so that the scheduling algorithms were disturbed as little as possible. However, the performance of the

template bundling algorithm should not be compromised to promote ease of implementation. Chapter 4 will present the results of the template bundling algorithm.

CHAPTER 4

EXPERIMENTAL RESULTS

This chapter presents the numerical data gathered for this thesis. The focus will be on how the new path with the template bundling algorithm, *SZU*, compares against the previous implementation that was used by the IMPACT compiler, *SMH*. SMH was based on the old path of the compiler using the machine description and bundling code given to the IMPACT research group by Intel. SMH also took a two-phase approach (scheduling/compacting) to the bundling problem. The three major differences are: (a) SMH assumes that the templates of the cycle are chosen based on the operations to be scheduled rather than as they are being scheduled, (b) SMH can adjust the level of compaction during the second phase and fine-tune code size reduction, and (c) SMH uses the Intel machine description instead of the one developed for the IMPACT compiler. Certain previous publications [11] of Itanium results by the IMPACT research group were based on the SMH implementation. The effect of the compaction phase of the new template bundling algorithm, SZU, will also be examined. The version without the compaction phase will be referred to as *SZU-NC*.

The results on the SPECint2000 benchmark suite are presented, excluding 252.eon since the IMPACT compiler did not handle C++ at the time of the experiments. All of the experiments begin from the same point in the IMPACT compilation process. Almost all of the

ILP optimizations, such as super/hyperblock formation, loop optimization, critical path reduction, have all been performed already. Scheduling, physical register allocation, and peephole and machine specific optimizations remain. The SPECint2000 numbers were generated on a Hewlett-Packard zx6000: two 1 GHz/3 MB L3 Itanium 2 processors, 8 GB RAM, Linux 2.4.2.1-gspec. The term *gspec* means that the operating system was enhanced to support general speculation, used to recover from mis-speculation. Pointer analysis and modulo scheduling were performed on all of the benchmarks except 253.perlbmk; the pointer analysis at the time of the experiments could not handle that one benchmark. The 176.gcc benchmark required a small manual fix; missing memory aliasing information due to errors in the pointer analysis allowed the scheduler to perform a bad reordering of operations that had to be corrected.

4.1 Performance Analysis

Table 4.1 shows the SPECint2000 performance results for SZU, SZU-NC, and SMH. Overall the differences in performance are slight. This is to be expected since all three compilations share the same starting point, after most of the ILP and performance enhancing transformations are already done, and the same scheduling algorithms. Therefore, the difference in performance comes from the different decisions made when forming the bundles and templates and the effect those decision have on peephole, machine specific optimizations.

The current compaction algorithm does not alter the cycle an operation is scheduled in, so changes in performance between SZU and SZU-NC would only come from changes in instruction cache behavior and microarchitecture front end issues only. The benchmark 186.crafty

Table 4.1 SPECint2000 results

Benchmark	SZU	SZU-NC	SMH	SZU/SZU-NC	SZU/SMH
164.gzip	754	754	728	1.000	1.036
175.vpr	680	677	664	1.004	1.024
176.gcc	775	769	822	1.008	0.943
181.mcf	337	340	337	0.991	1.000
186.crafty	713	679	687	1.050	1.038
197.parser	550	551	550	0.998	1.000
252.eon	X	X	X	X	X
253.perlbmk	664	669	652	0.993	1.018
254.gap	584	571	580	1.023	1.007
255.vortex	1181	1180	1187	1.001	0.995
256.bzip2	714	688	710	1.038	1.006
300.twolf	898	886	904	1.014	0.993
GEOMEAN	684.4	677.1	680.9	1.011	1.005

is known to suffer from instruction cache issues, and the SZU to SZU-NC ratio shows good performance improvement. The differences between SZU and SMH are harder to quantify. However, a closer examination showed that for the 176.gcc benchmark the SZU version spends a lot more time in the kernel than SMH, decreasing its performance. Wild loads due to mis-speculation is the cause of this phenomenon. The new template bundling algorithm allows more aggressive promotion of operations by finding more opportunities to break dependences and to reorder operations, creating more speculative instructions. This can sometimes have a detrimental effect if speculative operations fail rather than succeed, especially potentially excepting instructions (PEI) like memory load operations. Hopefully better memory analysis information or heuristics will minimize these effects.

Table 4.2 Backend component compilation time

Benchmark	SZU	SZU-NC	SMH	SZU/SZU-NC	SZU/SMH
164.gzip	5	5	7	1.00	0.71
175.vpr	31	30	38	1.03	0.81
176.gcc	237	236	308	1.00	0.77
181.mcf	2	1.5	2	1.33	1.00
186.crafty	53	52	65	1.02	0.82
197.parser	12	12	15	1.00	0.80
252.eon	X	X	X	X	X
253.perlbmk	41	41	71	1.00	0.58
254.gap	46	45	68	1.02	0.68
255.vortex	61	60	94	1.02	0.65
256.bzip2	5	5	7	1.00	0.71
300.twolf	274	277	283	0.99	0.97

4.2 Compilation Time Analysis

The compilation times presented in Table 4.2 are the number of minutes it takes for the IMPACT compiler to run from the Lcode internal representation through a series of SSA optimizations and through the scheduler, finishing with the final assembly code (HS to HSX to HSX_s in the IMPACT compilation process).

Except for 181.mcf, where the compilation time is very small already, the compilation times for the new template bundling algorithm, SZU, are all less than for SMH. This occurs even though SZU is an exhaustive search and always unschedules all operations in the issue group of the target cycle before trying to schedule. Section 6.1 presents possible extensions to further reduce the compilation time.

As shown by the SZU versus SZU-NC ratio, the compaction phase of SZU does not significantly extend the compilation time. Again, 181.mcf is an anomaly due to its short baseline compilation time. If the compaction phase is enhanced in the future the compilation time of

Table 4.3 Static NOP operations

Benchmark	SZU	SZU-NC	SMH	SZU/SZU-NC	SZU/SMH
164.gzip	3992	5849	3839	0.68	1.04
175.vpr	17525	22137	17341	0.79	1.01
176.gcc	190916	268709	186712	0.71	1.02
181.mcf	986	1547	970	0.64	1.02
186.crafty	17881	26883	17185	0.67	1.04
197.parser	12474	17034	12023	0.73	1.04
252.eon	X	X	X	X	X
253.perlbmk	92732	148601	91202	0.62	1.02
254.gap	89918	127836	87926	0.70	1.02
255.vortex	60623	75320	59703	0.80	1.02
256.bzip2	3494	4708	3359	0.74	1.04
300.twolf	29461	42511	29035	0.69	1.01

the compaction phase may increase. However, if the compaction phase remains a linear pass through the issue groups as it is now, it will most likely remain an insignificant portion of the compilation time.

4.3 NOP Analysis

Table 4.3 shows the number of static NOP operations present in the assembly code of each compilation process. Since most of the ILP and performance enhancing transformations are already done, the number of useful operations is approximately equal across the three compilations. In other words, the difference in code size comes from the difference in NOP operations. Although the difference in static NOP operations between SZU and SMH is relatively minor, SZU-NC presents major savings. This suggests that compressed templates are important to code size reduction.

Figure 4.1 shows the breakdown of dynamic operations between explicit NOP operations, predicate squashed operations, and useful operations for each compilation process across the benchmark suite. The results are normalized to the number of dynamic operations for the SZU compilation. For SZU versus SMH, there is no clear correspondence between the percentage of dynamic NOP operations and the overall performance of the benchmark (Table 4.1). In fact, for most of the benchmarks the number of explicit NOP operations between SZU and SMH is very similar. The compaction phase in SZU does contribute to reducing the number of explicit NOP operations. Again, there is not a significant performance benefit in reducing the number of NOP operations. Only programs that have decreased performance due front-end problems would benefit from this reduction in code size.

For two of the benchmarks, gzip and bzip2, SMH executed significantly less NOP operations than SZU despite a mere four percent difference in static NOP operations (Table 4.3). Although the performance of SZU was actually better (Table 4.1), this is still cause for investigation. Upon closer examination, it becomes evident that SMH achieves better code compression than SZU in certain cases. Although the number of cycles required for the control block is the same between SZU and SMH, the SZU compilation added an extra bundle, or three NOP operations. The effect becomes pronounced in the dynamic behavior because the control blocks where this occurred have high execution counts.

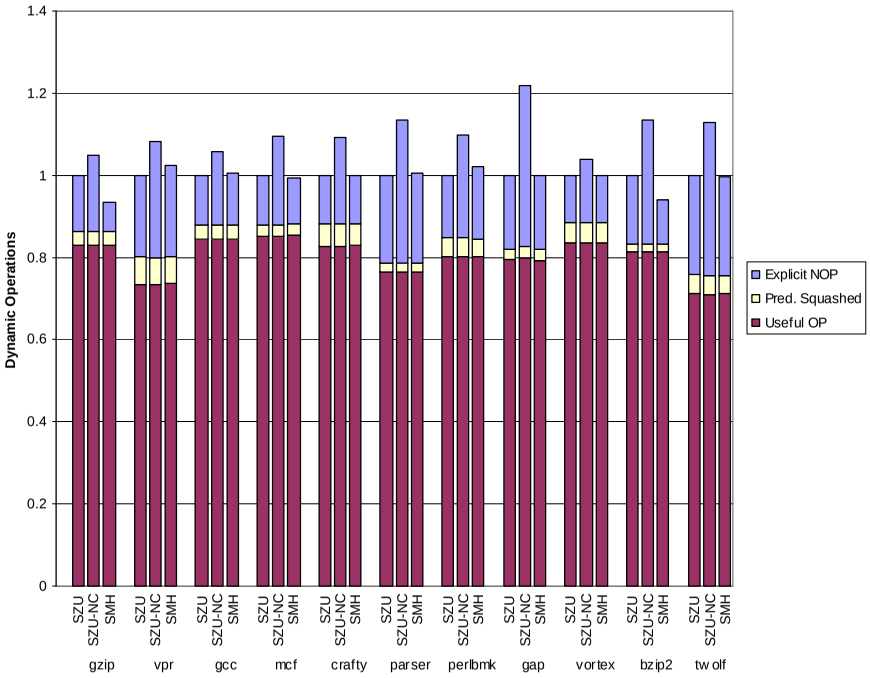


Figure 4.1 Dynamic operation accounting

CHAPTER 5

RELATED WORKS

This chapter goes over other schedulers for the IPF architecture in the literature. The focus will be on how the new constraints IPF introduced, such as syllable types and templates, are handled. Section 5.1 covers the ORC implementation, an open source compiler for IPF. Section 5.2 goes over a proposed implementation using integer linear programming. All other works are grouped together in Section 5.3.

5.1 ORC Implementation

The EPIC Open Research Compiler (ORC) published their work on resource management and template bundling for the Itanium implementation of IPF [3]. Their implementation utilized a finite-state automaton (FSA) for machine resource modeling, which was extended to encompass the constraints IPF introduced. The primary result showed that templates must be considered when scheduling for IPF.

The scheduler was separated into two parts, called the *high-level scheduler* and the *micro-level scheduler*. The high-level scheduler handled duties similar to the scheduling algorithm in this thesis while the micro-level scheduler encompassed the scheduling manager, the template bundling algorithm, the resource manager, and the machine description. The FSA was

function-unit-centric, keeping track of what issue ports are being used in the cycle. As additional operations are scheduled, the FSA advances from state to state according to the new issue ports being marked as used. Intracycle dependences were handled by checking the list of valid templates at the new state; at each state a list of valid templates is kept, based on the state transitions and operations scheduled to reach that state. If a valid order is not present, the algorithm reverts back to the previous state of the FSA. The ORC algorithm did not have a compaction phase. Rather, the micro-level scheduler kept a window of two cycles for template selection, thereby utilizing compressed templates. Template selections were finalized when the window moves on. For cyclic scheduling however, the ORC algorithm finalized template assignments after all of the operations for the loop have been scheduled. Results for one-cycle template selection, which would correspond to SZU-NC in this thesis, were also presented in the paper.

It is unclear how dispersal rules were handled in the ORC compiler. In other words, it is unclear how an operation's assignment to a specific function unit and how the effect of operation ordering were modeled. One possibility is to consider the dispersal rules while performing the intracycle dependence check, since the operation ordering in the cycle is considered at that time. Dispersal rules may not have had as much significance for the ORC implementation at the time of the paper since dispersal rules for the Itanium implementation were much simpler than Itanium 2. Not only did the Itanium have fewer function units to consider and a lesser variety of groupings of issue ports for instruction subtypes, operations with a superset syllable

had to dispatch to an issue port of the syllable type of the slot, i.e. an A-instruction type operation scheduled in an I-syllable slot will only disperse to an I-issue port and not an M-issue port like in Itanium 2.

5.2 ILP-Based Instruction Scheduling

Integer linear programming (ILP) has been proposed to perform instruction scheduling for IPF [12]. It is a mathematical and theoretically optimal scheduling technique. However, it is much more complex and expensive from a compilation standpoint. The algorithm presented takes a two-phase approach. A *macro-scheduling* phase assigns each instruction to an instruction group so that all instructions from one group can be executed simultaneously in a single clock cycle. This is done while ignoring bundling restrictions. A *bundling* phase is responsible for forming bundles that conform to the set of valid templates. All of this is done while representing the resource and template constraints in ILP. The work was further extended to show how speculation and cyclic code motion can be handled [13].

This thesis utilizes a bit-vector representation for machine resources. Although it may not be as mathematically optimal as ILP, it is easier to implement, and it can easily handle many different situations, such as cyclic code motion and software pipelining, and large programs such as the SPEC2000 benchmark suite. The template bundling algorithm implemented upon this representation has also been shown to be very time efficient.

5.3 Other Approaches

In Section 3.4, one of the alternatives mentioned is to integrate the scheduling algorithm when performing scheduling and template decisions. In particular, this combination should be able to take advantage of slack in the schedule to produce more compact code. This approach, integrating the operation and cycle selection with the template selection, was proposed in [14]. The emphasis was on improving code density as much as possible by performing template selection and operation scheduling simultaneously. A recursive algorithm was utilized to perform an exhaustive search to find the optimal solution. Nonoptimal heuristics were then introduced to help control compile time.

Although the paper does address certain cases where a NOP operation would exist under the current implementation of the template bundling algorithm, the results gathered for this thesis suggest that reduction of NOP operations does not directly translate into improvement in performance. Performance will improve from NOP operation reduction only if the application has instruction fetch problems, like 186.crafty. Furthermore, the algorithm presented in [14] only guaranteed minimum cycle code for a nonconstrained machine while arguing that code density is proportional to performance. On real hardware, trade-offs will have to be made in an intelligent manner between all of the operations that are ready to be scheduled. The implementation proposed by this thesis allows scheduling algorithms to be conceived and experimented separately from the template bundling concerns. Code density also appears to be a secondary effect on performance, being a major concern only in programs experiencing instruction cache

issues. Other methods for utilizing scheduling slack to enhance code density will be discussed in Section 6.2.

CHAPTER 6

FUTURE WORK

The first job of the scheduler of the compiler is to not make mistakes when generating a schedule for execution. Its job is even more important for in-order architectures like EPIC and IPF since there is no hardware support for dynamic reordering of operations. The template bundling algorithm framework presented in this thesis does this by handling the various restrictions in a cohesive intelligent manner. However, there is always room for improvement. After all, a scheduler that does not make mistakes can simply be one that schedules for maximum latency between all operations, one operation per cycle, or one that takes infinite time to ponder all possibilities. This chapter discusses how the template bundling algorithm can be improved, focusing on compilation time and code density.

6.1 Compilation Time Improvements

An extension to the current template bundling algorithm to further reduce compilation time is to insert the new operation x into the issue group g without unscheduling all of the operations Y currently scheduled in g (Figure 3.3). If the insertion is successful, then the run time would be a factor of ITR rather than exponential. However, as appealing as this improvement sounds, further study is necessary. Regardless of the insertion algorithm's implementation, it will likely not be an exhaustive search like the current template bundling algorithm, since its starting point

is constrained. Therefore, it is conceivable that the insertion algorithm may fail in a situation that contains a valid schedule if an exhaustive search is performed. Consequently, the benefit of the insertion algorithm needs to be balanced against the probability of failure, which would invoke the exhaustive search template bundling algorithm.

An insertion algorithm is nontrivial, due to dispersal rules, templates, and NOP operations. The template bundling algorithm schedules NOP operations in slots without real operations because they consume resources and consequently affect the dispersal of real operations. In other words, in order to insert a new operation x into g, it must take the place of an existing NOP operation. However, this replacement will have effects on the dispersal rules and may even require the changing of template assignments.

One approach to the insertion algorithm is to require x to take the place of a NOP operation, n, without changing the placement of any of the existing real operations Y or the templates of the bundles in g. However, this means the NOP operation's slot must have a syllable type compatible with x. Furthermore, the issue port currently consumed by n must also be what x would disperse to according to the dispersal rules. If the second condition is not satisfied, any change will percolate down to the succeeding operations and possibly alter what issue ports they disperse to, producing split issues.

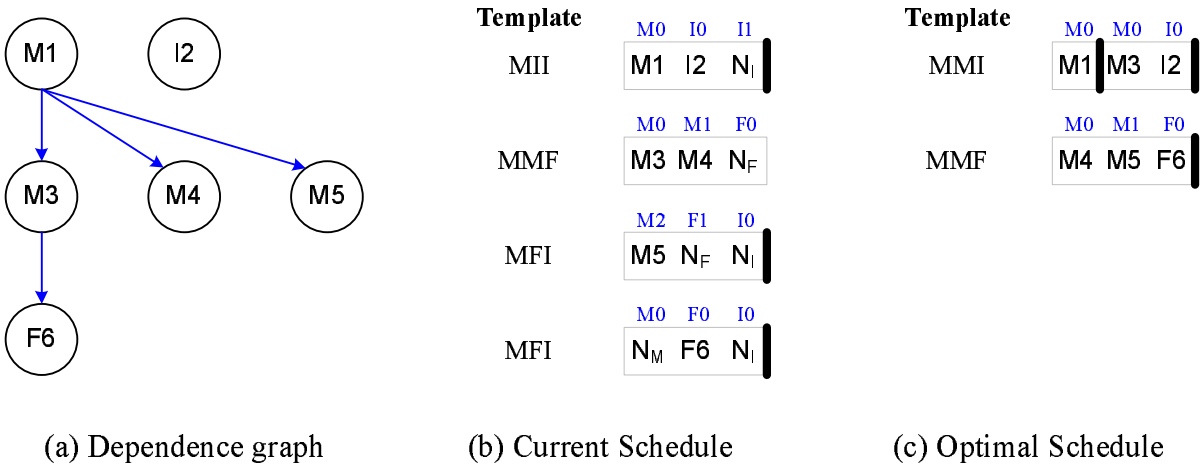
Other approaches include limited updates to the template assignments, modifications to the placement of the existing operations Y, or a combination of both. However, the insertion algorithm will then begin to resemble the exhaustive search template bundling algorithm and gain will lessen as complexity increases. It is my opinion that it would be best to implement

the previous approach, guided by heuristics that dictate when to skip the insertion algorithm and go straight to the exhaustive search algorithm instead. The heuristics can be based on the number of operations already scheduled in g, $|Y|$, the syllable type of x, the existing templates, etc.

Finally, more improvements in compilation time may be possible by trimming down the search space of the exhaustive search. The current implementation of the template bundling algorithm is made to be general and adaptable. Machine-specific restrictions could be integrated to constrain the search space. There may also be other more general, nonmachine specific bounds not yet discovered.

6.2 NOP Operations Analysis and Improvements

Since the framework follows the IPF definition that NOP operations disperse and affect resource consumption, their presence cannot be treated lightly. The template bundling algorithm properly handles NOP operations by scheduling them into the issue group. However, reduction of NOP operations will improve code density and performance in certain cases. Although the results indicate that the impact on performance from reducing NOP operations is not dramatic (see SZU versus SZU-NC results in Chapter 4), it cannot be ignored. Due to the requirement of forming bundles with valid templates and the limited selection of template types, arguably NOP operations will always be present in IPF code. Although the compaction phase does a good job in reducing the number of NOP operations, it is by no means optimal.



(a) Dependence graph (b) Current Schedule (c) Optimal Schedule

Figure 6.1 Optimal NOP operation example

Figure 6.1 is a motivating example illustrating the deficiency of code density in the current template bundling and compaction algorithm. Figure 6.1(a) is a dependence graph showing the operations and the dependences between them. Since Instr-I2 does not have any other operation dependent upon it, it can be scheduled in any cycle. The critical path consists of Instr-M1, M3, and F6. The scheduling algorithm currently in use attempts to schedule an operation as soon as it is ready, resulting in the schedule shown in part (b). The schedule with optimal code density is shown in Figure 6.1(c), eliminating a total of six NOP operations. The scheduler delays Instr-I2, M4, and M5 by one cycle each from when the operation is ready to be executed to maximize code density.

In order to achieve the schedule shown in part (c), the scheduler has to realize that operations off of the critical path not only can be scheduled later, but should be scheduled later to achieve better code density. One approach in the current framework would be to enhance the compaction phase to consider scheduling slack while compacting. Currently, the compaction

phase does not allow operations to cross cycle boundaries. Since the scheduling algorithm tends to pull up operations by scheduling them as soon as they are ready, the compaction phase can try to exploit slack in the schedule by delaying certain operations to enhance code density. Rather than start the compaction phase from the top of the control block, it can take a bottom-up approach instead. When the compaction algorithm is considering compacting two issue groups, allow operations in the preceding issue group with slack in its schedule to be pushed down into the later cycle. Heuristics will be needed to judiciously select which operations to delay and judge their effect. Scheduling slack can also be utilized to perform other optimizations other than reduction of NOP operations.

CHAPTER 7

CONCLUSION

IPF, the first hardware implementation of an EPIC architecture, introduced new concepts to the compiler's scheduler. These new concepts, bundles, syllables, templates, dispersal rules, etc., are a continuation in the evolution of operation dispersal in computer architecture. When the responsibility of tracking operation dispersal was transferred from the hardware of out-of-order architectures like superscalar machines to the compiler of in-order architectures like VLIW and generic EPIC machines, the hardware dispersal mechanism became exempted from much of the decision making. However, it still needed to support symmetrical, cross-bar-like dispersal of operations from slots to functional units. These new IPF concepts give more power and responsibility to the compiler and simplify the operation dispersal in hardware even further. They simultaneously constrain operation dispersal away from the symmetrical model assumed previously and give the compiler a limited ability to configure the operation dispersal mechanism as it sees fit.

This thesis presents extensions to the scheduler of an EPIC compiler, the IMPACT research compiler, to handle these new constraints introduced by IPF. A modular implementation, governed by the template bundling algorithm in the scheduling manager, these extensions not only handle the new concepts introduced by IPF but do so in a way that can be expanded and extended upon for future architectures should they employ similar constraints. The separation of

responsibilities also allow experimentations with different scheduling algorithms, handling the machine specific template bundling scheduling concerns separately. Despite being an exhaustive search, the template bundling algorithm has proven to be efficient, improving compilation time from a previous implementation.

The work done by this thesis lays the foundation for future exploration in scheduling for IPF. Possibilities include performance and compilation time oriented research. It provides a quick, solid scheduling platform for experimentations with scheduling algorithms and schedule time transformations. There is also still room for improvement within the template bundling algorithm for compilation time and NOP operation reduction.

REFERENCES

[1] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Volume 1: Application Architecture*, Document Number 245317-003, December 2001.

[2] J. C. Gyllenhaal, "A machine description language for compilation," M.S. thesis, University of Illinois at Urbana-Champaign, 1994.

[3] D.-Y. Chen, L. Liu, C. Fu, S. Yang, C. Wu, and R. Ju, "Efficient resource management during instruction scheduling for the EPIC architecture," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2003, pp. 36–45.

[4] Intel Corporation, *Intel Itanium 2 Processor Reference Manual for Software Development*, Document Number 251110-001, June 2002.

[5] Intel Corporation, *Intel Itanium Processor Reference Manual for Software Development*, Document Number 245320-003, December 2001.

[6] J. A. Fisher, "Global code generation for instruction-level parallelism: Trace scheduling-2," Hewlett-Packard Laboratory, Tech. Rep. HPL-93-43, June 1993.

[7] R. A. Bringmann, "Compiler-controlled speculation," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1995.

[8] B. R. Rau, "Iterative modulo scheduling," *International Journal of Parallel Processing*, vol. 24, pp. 3–64, February 1996.

[9] J. C. Gyllenhaal, B. R. Rau, and W. W. Hwu, "HMDES version 2.0 specification," IMPACT, University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-96-03, 1996.

[10] J. C. Gyllenhaal, "An efficient framework for performing execution-constraint-sensitive transformations that increase instruction-level parallelism," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, 1997.

[11] J. W. Sias, M. C. Merten, E. M. Nystrom, R. D. Barnes, C. J. Shannon, J. D. Matarazzo, S. Ryoo, J. V. Olivier, and W. W. Hwu, "Itanium performance insights from the IMPACT compiler," in *Hot Chips 13*, August 2001.

[12] D. Kästner and S. Winkel, "ILP-based instruction scheduling for IA-64," in *Proceedings of the Workshop on Languages, Compilers and Tools for Embedded Systems LCTES-01*, vol. 36, June 2001, pp. 145–154.

[13] S. Winkel, "Optimal global scheduling for Itanium™processor family," in *Proceedings of the 2nd Annual Workshop on Explicitly Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques*, November 2002, pp. 58–69.

[14] S. Haga and R. Barua, "EPIC instruction scheduling based on optimal approaches," in *Proceedings of the 1st Workshop on Explicitly Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques*, December 2001, pp. 23–32.