JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Register Allocation on the Intel® Itanium® Architecture

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

TECHNISCHEN WISSENSCHAFTEN

Eingereicht von:
Dipl.-Math. Gerolf Fritz Hoflehner

Angefertigt am:
Institut fuer Systemsoftware

Beurteilung:
o.Univ.-Prof. Dr. Dr.h.c. Hanspeter Mössenböck (Betreuung)

a.Univ.-Prof. Dr. Andreas Krall

San Jose, März 2010

Abstract

Register allocators based on graph-coloring have been implemented in commercial and research compilers since Gregory Chaitin's and colleagues pioneering work in the early 1980's. A coloring allocator decides which live range (a program or compiler generated variable) is allocated a machine register ("allocation problem"). The Itanium processor architecture supports predicated code, control- and data speculation and a dynamic register stack. These features make the allocation problem more challenging. This thesis analyses and describes efficient extensions in a coloring allocator for the Itanium processor.

- **Predicated code**: The thesis describes compile time efficient coloring methods in the presence of predicated instructions, without compromising run-time performance of a more elaborate allocator based on the predicate query system, PQS. In particular it classifies predicated live ranges and shows that classical register allocation techniques can be used effectively to engineer efficient coloring allocators for predicated code. When predicated code is generated from compiler control flow more expensive predicate analysis frameworks like PQS don't have to be employed.
- **Speculated code**: The thesis describes a new method of efficiently allocating speculated live ranges avoiding spill code generated by a more conservative method. In particular the NaT propagation problem is solved efficiently.
- **Dynamic register stack**: The thesis reviews methods to use the dynamic register stack effectively in particular regions with function calls and/or pipelined loops.
- **Scalable allocation**: A generic problem of coloring allocators is that they can be slow for large candidate sets. This thesis proposes the scalable allocator as a generic coloring method capable of allocating effectively programs with large register candidates sets. The methods can also be used for parallel allocation e.g. on multi-core machines.

The experimental results on the CPU2006 benchmark suite illustrate the effectiveness of new methods. Finally, the thesis reviews the development of coloring allocators since Chaitin.

Kurzfassung

Gregory Chaitin und seine Kollegen haben um 1980 die Pionierarbeit für Registerallokation basierend auf Graphfärbung („Farballokator“) geleistet. Diese Allokatoren entscheiden, welchen „Lebensspannen“ (d.h. deklarierte oder vom Compiler erzeugte Variablen, engl. live ranges) Maschinenregister („Farben“) zugeteilt werden (Allokationsproblem). Der Itanium-Prozessor unterstützt Instruktionen mit Prädikaten (bedingt ausführbare Instruktionen), Kontroll- und Datenspekulation sowie einen dynamischen Register Stack. Diese Eigenschaften erschweren die Lösung des Allokationsproblems. Die vorliegende Dissertation untersucht und beschreibt effiziente Erweiterungen in einem Farballokator für den Itanium-Prozessor.

- **Prädikate**: Es werden effiziente Methoden (zur Übersetzerzeit) für Farballokatoren vorgestellt, die Lebenspannen in Code mit bedingt ausführbaren Instruktionen allokieren. Insbesondere werden „prädikatierte“ Lebensspannen klassifiziert und es wird gezeigt, das klassische M ethoden zu einem effizienten Farballokator für diese Lebensspannen erweitert werden können. In einem Compiler kann die Allokation mit diesen Methoden genau so effizienten Code generieren wie aufwendigere Verfahren, insbesondere Verfahren, die das „predicate query system“ (PQS) benutzen.
- **Spekulation**: Es wird eine neue Methode erläutert, die – im Vergleich zu konservativen Verfahren – Spill Code für spekulative Lebensspannen vermeiden kann. Inbesondere wird eine effiziente Lösung für das NaT Propagation Problem vorgestellt.
- **Dynamischer Register Stack**: Es wird beschrieben, wie der dynamische Register Stack in Code mit Funktionsaufrufen oder „pipelined“ Schleifen (engl. „software-pipeline loops“) effizient verwendet werden kann.
- **Skalierbare Allokation**: Es wird der skalierbare Allokator vorgeschlagen für die Lösung Allokationsprobleme beliebiger Grösse. Skalierbare Allokation erlaubt insbesondere die Parallelisierung des Allokationsproblems und ist unabhängig von der Prozessor-Architektur.

Die experimentellen Resultate für die CPU2006 Benchmark Suite zeigt die Effizienz der vorgestellen Verfahren. Schließlich enthält diese Dissertation einen ausführlichen Überblick über die Forschungsergebnisse für Farballokatoren seit Chaitin.

Acknowledgements

The tree with our fruits is watered by many. First, this work would not have been possible without Intel and many of its excellent engineers. Roland Kenner implemented the original version of the graph-coloring allocator and PQS. The idea to eliminate interferences for portions of a data speculated live range (Section 7.2) had been implemented in the compiler before 1999, the year I started at Intel. Daniel Lavery was my exceptional engineering manager for seven years, taking a lot of interest in and continuously challenging my thoughts on register allocation. His questions and curiosity helped shape the section on predicate-aware register allocation. I had the good fortune to benefit from many discussions and working with an outstanding team and colleagues: Howard Chen, Darshan Desai, Kalyan Muthukumar, Robyn Sampson and Sebastian Winkel. Alban Douillet and Alex Settle helped bringing to life the product version of the multiple alloc algorithms during their summer internships. The Intel Compiler Lab provided a great place to work and let me drive innovation amidst the challenges of tight product schedules.

Second, I would like to thank Prof. Hanspeter Mössenböck for his patience and support. His kind personality, openness, and advice were inspiring. I cannot imagine how he could have been more supportive.

Finally, and most importantly, I thank the woman in my life, Elisabeth Reinhold. Her push and loving support were always there when I needed it most.

Contents

1 Introduction

1.1 Compilers and Optimizations

Optimizing compilers can be thought of a 3-stage process, which transforms a source code program into a linkable object file for a target machine: in the first stage, the front-end translates a source language like C or Fortran into an intermediate representation. The second stage, the optimizer, applies a set of local and global transformations (=optimizations) with the goal to speed-up run-time performance of the final executable. Local transformations work on a sequence of branch-free statements. Global transformations gather information about e.g. expressions and variables in the entire routine using dataflow analysis. Classical optimizations like partial-redundancy elimination, common sub-expression elimination, dead code elimination, or loop-invariant code motion apply this information. Optimizers may apply also loop transformations like loop unrolling, loop splitting, loop fusion or software pipelining. More aggressive optimizers gather interprocedural optimizations and perform optimizations like procedure inlining. Finally, the third stage, the code generator (=back-end), translates the intermediate representation produced by the optimizer into object code. Typically, the final stage involves several phases including instruction selection, instruction scheduling and register allocation.

1.2 Register Allocation based on Graph-Coloring

A register is the fastest memory location in a CPU. Each CPU has a limited set of registers. During program execution registers hold the values of program variables or compiler temporaries. In general an optimizing compiler performs optimizations under the assumption that an infinite number of registers is available in the target machine. Thus optimizations are register pressure unaware, which simplifies their design. Pressure refers to the fact that machine register resources are limited. It is the job of the register allocator to map the symbolic registers in the intermediate representation of the compiler to actual registers of the target machine. A graph-coloring based register allocator abstracts the allocation problem to coloring an undirected interference graph with K colors, which represent K machine registers. The nodes in the graph are the symbolic registers, and two nodes are connected by an edge when they cannot be assigned the same register. As a rule the program executes faster the more symbolic registers can be allocated to machine registers.

1.3 Itanium Processor Family

The Itanium processor family -or IA-64- is a commercially available implementation of the EPIC ("Explicitly Parallel Instruction Computing") computing paradigm. In EPIC the compiler has the job of extracting instruction level parallelism (ILP) and communicating it to the hardware. Itanium enhances concepts usually seen in VLIW processors. The long instruction words are fixed-size bundles that contain three instructions (operations). It is a 64-bit computer architecture distinguished by a fully predicated instruction set, a dynamic register stack, rotating registers and support for control- and data speculation. Predication and speculation allow the compiler to remove or break two instruction dependence barriers: branches and stores. With predicates the compiler can remove branches ("branch barrier removal"), with control speculation it can hoist load instructions across branches ("breaking the branch barrier"), and with data speculation it can hoist load instructions across stores ("breaking the store barrier"). Using predication and rotating registers the compiler can generate kernel-only pipelined loops. The dynamic register stack gives the compiler fine-grain control over stacked register usage. In general exploiting instruction-level parallelism using Itanium features increases register pressure and poses new challenges for the register allocator.

1.4 Overview

This thesis describes extensions of graph-coloring based register allocation methods that exploit the distinguishing IA-64 features. The methods have been implemented in the Intel Itanium production C/C++/Fortran compiler and are hardware specific. Orthogonal to the Itanium specific methods is the scalable allocator, which is applicable to other architectures and/or compilation environments. It can address compile time problems for programs with large candidate sets, e.g. some generated programs in server applications, and demonstrates how allocation can be parallelized. The rest of the thesis is organized as follows: The first three sections develop background. Chapter 2 gives an overview of the

IA-64 (micro-) architecture emphasizing aspects relevant for register allocation. Chapter 3 reviews register allocation literature since Chaitin's seminal work. Chapter 4 takes a look at the major code generator phases in the Intel Itanium compiler. The following three sections cover IA-64 specific allocation techniques: Chapter 5 discusses register stack allocation. Chapter 6 gives an in-depth discussion of predicate-aware register allocation. Chapter 7 describes allocation for control- and data-speculated code. Chapter 8 proposes scalable register allocation that addresses compile time issues for large register candidate sets and is a method for parallelizing coloring allocators. This chapter is general and not specific to Itanium. Chapter 9 discusses related work with respect to the core contributions of this thesis. Chapter 10 has implementation and experimental results. Chapter 11 concludes the thesis and lists future work. The Appendix has an Itanium assembly code example, reviews the concept of irreducible control flow graphs and code for PQS query functions.

2 Background on IA-64

Fundamental for any computer architecture is the instruction set architecture (ISA). The first section gives a high-level survey of Itanium instructions. The goal is to provide sufficient background knowledge for reading basic Itanium assembly code. The remainder of this section focuses on the aspects of Itanium that are relevant for register allocation, including register files, register classes, register stack, predicated and speculated code. In general, the sections cover material at the architecture level. The term architecture specifies how to write or compile semantically correct Itanium programs. Only the discussion of the register stack will involve a (high-level) description of the micro-architecture. The micro-architecture specifies how the Itanium processor actually implements an architectural feature. Knowledge of the micro-architecture enables the compiler (or assembler writer) to generate faster programs. However, micro-architecture details may change from one processor generation to the next. In this case, programs relying on micro-architectural details must be re-compiled (or re-tuned) for the newer generation to achieve best possible run-time performance. The basic references for the background material are the Intel manuals [42][43][44]. Winkel [76] has an overview of the IA-64 ISA and instruction dispersal rules, which we don't discuss.

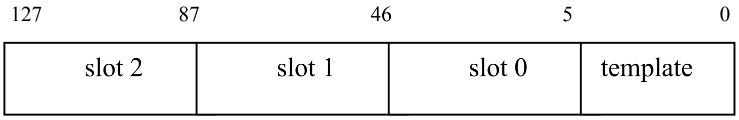
2.1 IA-64 Instructions

IA-64 instructions are grouped into bundles. A bundle is a simple 128 bit structure that contains three 41 bit instruction fields ("slots") and a 5 bit template that describes the execution unit resource each instruction requires. The template bits can also specify the location of a stop bit, which delimits instructions that can execute in parallel. In general instructions can execute in parallel as long as there are no read after write (RAW) or write after write (WAW) dependencies between instruction operands. A set of instructions that could execute in parallel is an instruction group. Two instructions with a write after read (WAR) dependence can be contained in the same instruction group. Per cycle Itanium can execute up to 6 instructions or two bundles in parallel, although instruction groups may contain any number of instructions. The task to extract instruction level parallelism, forming instruction groups and grouping instructions in bundles is with the compiler. Instructions between two stop bits are in the same instruction group.

Instruction:

[(qp)] mnemonic.[*compl*]* dest=src

Bundle:



Instruction Group:

…;; MFI MFI MLX ;; …

Figure 1 IA-64 Programming Building Blocks

Figure 1 illustrates the Itanium instruction format, bundles and instruction groups. “qp” is the qualifying predicate. It is encoded as a predicate register (see 2.3). It guards whether the instruction result is committed (“retired”) or not. An instruction is retired only when the qualifying predicate is set (=True). *Mnemonic* is a unique identifier (“opcode”) for an IA-64 instruction. *Compl* is a set of modifiers (“completers”) to the basic Mnemonic functionality. Completers are optional. An instruction may have no completer, one or more than one. *Dest* is a comma-separated list of output registers or a store address. *Src* is a comma-separated list of input registers, constants, or a load address. Specific examples of Itanium instructions are below.

Itanium instructions are grouped into types (Table 1). A type is a qualifier that suggests on which functional unit an instruction can execute. There are:

- Six instruction types: M-type, I-type, A-type, F-type, B-type, and LX-type
- Four types of execution units M-Unit, I-Unit, F-Unit, and B-unit
- 12 basic bundle template types MII, MI_I, MLX, MM I, M_MI, MFI, MMF, MIB, MBB, BBB, MMB, and MFB. A “_” in the bundle template type indicates a stop bit and is encoded in the template bits. There are only two template types that allow a stop in the middle of a bundle: M_MI has a stop bit after the first

instruction, and MI_I has a stop bit after the second instruction. A stop bit separates instruction groups. Therefore in M_MI the first M -type instruction is the last instruction of an instruction group, while the second M -type together with the I_type instruction is in a different instruction groups. Similar, in MI_I the M -type and the first I-type instruction are in the same instruction group, while the second I-type instruction is the first instruction of a new instruction group.

With the information about instruction types the hypothetical instruction group MFI MFI MLX in Figure 1 consists of eight instructions that could execute in parallel: three M -type, two F-type and two I-type instructions as well as one LX-type instruction.

Table 1 IA-64 Instruction Types and Execution Unit Types

Instruction Type	Description	Execution Unit Type
A-type	Integer ALU Instructions	Executes on I-unit or M-unit
I-type	Integer Non-ALU Instructions	Executes on I-unit
M-type	Memory Instructions	Executes on M -unit
F-type	Floating-point Instructions	Executes on F-unit
B-type	Branch Instructions	Executes on B-unit
LX-type	Extended (2 slot) Instructions	Executes on I-unit

A-type instructions can execute in a memory (M) unit or integer (I) unit. There is one LX instruction ("movl"). It consumes two bundle slots to move a 64bit constant into a register and executes on an I-unit. The Itanium-2 processor has four M [M0, M1, M2, M3], two F [F0, F1], two I [I0, I1], and three B [B0, B1, B2] execution units, which is sufficient to sustain a throughput of six instructions per cycle. Each of the 12 basic bundle templates may have a stop set after the third instruction, so Itanium supports 24 bundle template types. A future generation Itanium processor may define up to four new bundle types. This is determined by the 5 bits in the template bit field. The execution order of the instructions in a bundle is in-order proceeding from slot 0 to slot 2.

The remainder of this section gives specific examples for Itanium instruction. This will be sufficient to read small Itanium assembly programs like the example in Appendix 12.1 on p. 138). In all tables the instruction semantics is in C-style pseudo-code or informal. Subscripted letters r_i, p_i or f_i represent general, predicate and floating-point registers respectively.

Table 2 has examples for A-type instructions. They include arithmetical operations like integer “add”, logical operations like “and complement”, “or” and compare “cmp” instructions, where *crel* denotes a comparison relation. Examples for comparison relations are ‘eq’ (=”equal”), ‘ne’ (=”not equal”), or ‘gt’ (=”greater than”).

Table 2 A-type Instructions

	Instruction Type	Examples	
		Syntax	Semantics
1	A-type	(qp) add r_1=r_2,r_3	r_1=r_2+r_3
2	A-type	(qp) andcm r_1=r_2,r_3	r_1=r_2&~r_3
3	A-type	(qp) or r_1=r_2,r_3	r_1=r_2|r_3

4	A-type	(qp) cmp.*crel* p_1,p_2=r_1,r_2	qp=1 and r_1 *crel* r_2:
			p_1=1 p_2=0
			qp=1 and !(r_1 *crel* r_2):
			p_1=0 p_2=1

5	A-type	(qp) cmp.*crel*.unc p_1,p_2=r_1,r_2	qp=0:
			p_1=0 p_2=0
			qp=1 and r_1 *crel* r_2:
			p_1=1 p_2=0
			qp=1 and !(r_1 *crel* r_2):

6	A-type	(qp) cmp.*crel*.or p_1,p_2=r_1,r_2	qp=1 and r_1 *crel* r_2: p_1 = 1 p_2 = 1
7	A-type	(qp) cmp.*crel*.and p_1,p_2=r_1,r_2	qp=1 and !(r_1 *crel* r_2): p_1=0 p_2=0

Instruction 5 in Table 2 is an unconditional compare instruction. This instruction has two completers, *crel* for the actual comparison relation, and ‘unc’ to indicate the compare is unconditional. This means, that the two destination predicate registers, p_1 and $p_{2,}$ are initialized to zero even when the qualifying predicate is clear (=False). Unconditional compares are used to initialize predicates in if-converted code (Section 2.2). Remarkable also are parallel compares, like instruction 6 and 7: they set or clear both destination predicates when the qualifying predicate is set, depending on the compare type and result. Parallel compares are used to evaluate logical ‘or’ and ‘and’ expressions in parallel (within the same bundle or instruction group).

I-type instructions (Table 3) include bit manipulations like deposit (“dep”, instruction 8), extract (“extr”, instruction 9), arithmetical shifts (“shr” , instruction 12) and zero extension (“zxt”, instruction). Table 3 shows also the 64-bit move instruction, which has LX-type, consumes two slots in a bundle and executes on an I-Unit.

Table 3 I-type and LX-type Instructions

	Instruction Type	Examples	
		Syntax	Semantics
8	I-type	(qp) dep r_1=r_2,r_3,pos,*len*	Deposit bit field merges bit field of length *len* starting at bit 0 in r_3 with r_2 at bit position *pos*. The result of the merge is placed in r_1.
9	I-type	(qp) extr r_1=r_3,pos,*len*	Extract bit field of length *len* starting at bit pos in r_3, sign extend and store right-justified in r_1
10	I-type	(qp) chk.s r_1, target	Control speculation check: Branch to target when NAT bit is set in Register r_1.
11	I-type	(qp) mov r_1=pr	Read predicate registers and store in r_1
12	I-type	(qp) shr r_1=r_2,r_3	Arithmetic shift right: r_1= r_2>>r_3
13	I-type	(qp) zxt4 r_1=r_3	Zero extend value of r_3 above bit 31 and store in r_1
14	LX-type	(qp) mov r_1=*imm_64*	Move 64bit immediate value *imm_64* into r_1

M-type instructions (Table 4) include loads, stores and the alloc instruction, which manages the register stack. They also contain transfer instructions between floating-point unit and integer units. A *getf* instruction is used to move data from a floating-point register to an integer register, while a *setf* instruction transfer the value from an integer register into the 64 bit significant of a floating-point register. These transfers are necessary, since integer multiply and divide must be performed in a floating-point unit.

Table 4 M-type Instructions

	Instruction Type	Examples	
		Syntax	Semantics
15	M-type	(qp) ld8 r_1=[r_3]	Load 8 bytes into r_1 from address in r_3
16	M-type	(qp) ld4.s r_1=[r_3]	Control speculated 4 byte load
17	M-type	(qp) ld2.a r_1=[r_3]	Data speculated 2 byte load
18	M-type	(qp) ld1.sa r_1=[r_3]	Control and data speculated 1 byte load
19	M-type	(qp) st8 [r_3]=r_1	Store 8 byte content of r_1 at address in r_3
20	M-type	(qp) getf.sig r_1=f_2	Store 64bit significant of f_2 in r_1
21	M-type	(qp) setf.sig f_1=r_2	Store value of r_2 in significant of f_1.
22	M-type	alloc r_1=ar.pfs,i,l,o,r	See Figure 5.

F-type instructions (Table 5) are floating-point instructions. Itanium supports single, double and extended (82 bit) floating point operation. A fundamental building block of all floating-point operations is the ‘floating-point multiply and add’ instruction, *fma*.

Table 5 F-type Instructions

	Instruction Type	Examples	
NR		Syntax	Semantics
23	F-type	(qp) fma.s f_1=f_3,f_4,f_2	f_1=f_3*f_4+f_2 rounded to single precision (‘s’ completer).
24	F-type	(qp) xma.l f_1=f_3,f_4,f_2	f_1=f_3*f_4+f_2 as 64-bit integer operation. The low (‘l’ completer) 64-bit of the result are stored in f_1.
25	F-type	(qp) fadd.d f_1=f_2,f_3	f_1=f_2+f_3 rounded to double precision (‘d’ completer).

The fma instruction computes the product of f_3 and f_4 and adds f_2 in infinite precision, and rounds the final result to the format specified in the completer. The *xma* instruction computes $f_3 * f_4 + f_2$, where the FP registers are interpreted as 64 bit integers. The intermediate value of the product is 128 bit. This instruction is used to perform integer multiply.

Table 6 B-type Instructions

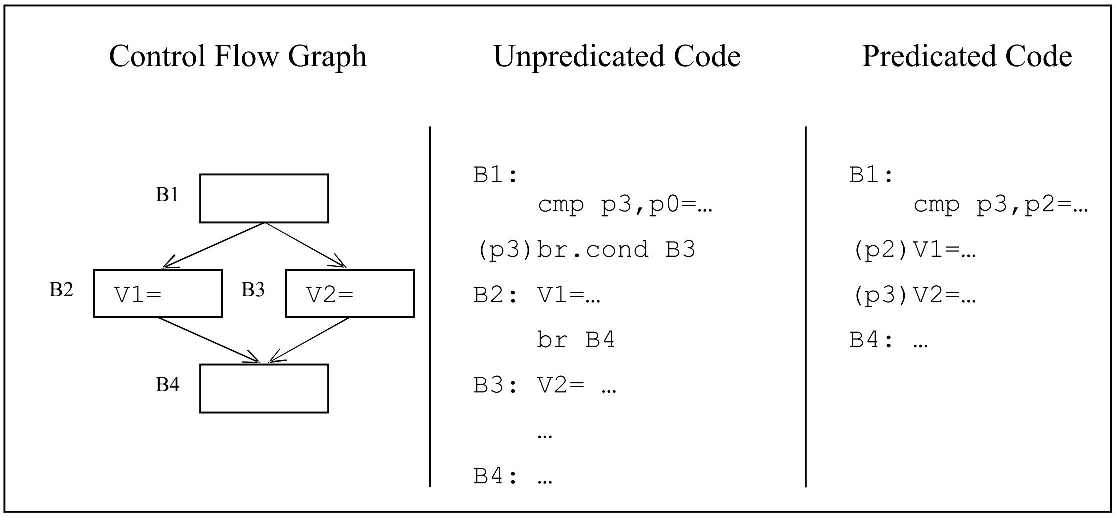
	Instruction Type	Examples	
		Syntax	Semantics
26	B-type	(qp) br.cond.*sptk.many* target	qp=1: IP-relative conditional branch. The condition is encoded in qp which is set in a separate cmp instruction.
27	B-type	(qp) br.call *foo*	qp=1: Invoke procedure *foo*
28	B-type	(qp) br.cloop target	qp=1 and loop count LC is not zero: Decrement LC and branch to target
29	B-type	(qp) br.ctop target	In software pipelined loops: branch to target if qp=1 and (LC != 0 or EC > 1), where LC is the loop count and EC the epilog count. In this context epilog count EC is the number of iterations that must finish before the pipelined loop can exit.
30	B-type	(qp) br.ret	qp=1: Procedure return
31	B-type	clrrrb	Clear register rename base registers

B-type instructions (Table 6) include IP-relative branches, calls, return, loop branches and clrrrb, which is used to rotating register bases for software pipeline loops. All branches can be conditional when a qualifying predicate is specified. They may have ‘whether’ completers like sptk, which the compiler issues when it statically predicts a branch is taken. The compiler can also specify whether the processor should fetch “few” or “many” bundles at the target address. The values for “few” and “many” are micro-architecture dependent.

2.2 Predication

Predication is the conditional execution of an instruction guarded by a qualifying predicate. On IA-64 the qualifying predicate is a binary (“predicate”) register that holds a value of 1 (=True) or 0 (=False). The (qualifying) predicate register is encoded in the instruction. When its value is 1 at run-time, the predicate is set. When the value is 0 at run-time, the predicate is clear. When the value of the predicate register is set, the instruction executes, potential exceptions get raised, and results are committed to the architectural state. When the value of the predicate register is clear, the instruction still executes, but no exception is raised and results are not committed. This means that the instruction “flows” through the instruction pipeline and gets discarded only in the last pipeline stage, the write-back (WRB) stage, even when the qualifying predicate of the instruction is clear. The default qualifying predicate is the constant predicate register p0, which is always set. An unpredicated instruction e.g. on a classical RISC architecture can be considered a special case of a predicated instruction which has its qualifying predicate always set. On IA-64 there are 64 predicate registers. Therefore the encoding of the qualifying predicate consumes 6 bits, which is one of the reasons why operations (instructions) are 41 bit wide. Another reason for the odd number of bits per instruction is the huge register file (see Section 2.3) with 128 general and 128 floating-point registers.

The predicate registers are written by compare instructions. A compare instruction has two target predicate registers which – in the case of conditional or unconditional compares - represent a true and false condition value at run-time. The typical consumer of a predicate register is a branch instruction. On IA-64 conditional branches are predicated branches. When the qualifying predicate of a branch is set at run-time, the branch is taken. When the qualifying predicate of the branch is clear, the branch is not taken and the instruction following (statically) the branch gets executed. On IA-64 almost all instructions are predicated. This means they may contain a qualifying predicate that is either set (=true) or clear (=false) at run-time. As (almost) fully predicated architecture IA-64 supports if-conversion. If-conversion is a compiler optimization that eliminates conditional forward branches and its associate branch miss-prediction penalty, if the branch is hard to predict by the hardware branch predictor. Branch miss-prediction penalty is the re-steer cost when the hardware branch predictor predicts a branch direction incorrectly. The instructions dependent on the branch are predicated up to a merge point in the original control flow graph. This eliminates the conditional branches and converting control dependencies into data dependencies [2]. As a result it transforms a control flow region to a linear ("predicated") code region. The paths in the control flow graph become execution traces in the predicated code. In the predicated region all paths of the original control flow region overlap. Therefore the processor supporting if-conversion must have sufficient resources to potentially execute any of N program paths, although at any given point in time only one actually executes. If-conversion is illustrated in Figure 2. It shows a simple if-then-else structure ("hammock"), the unpredicated code with branches a compiler generates without if-conversion, and the if-converted code. There are two paths in the original control flow graphs and correspondingly two execution traces in the if-converted code (one trace contains cmp – (p2) V1 … , the other trace cmp – (p3) V2=...). B4 in the original control flow serves is a merge point. All branches have been eliminated in the if-converted code. In case the conditional branch is mis-predicted in unpredicated code, the if-converted code is more efficient.



Unpredicated Code

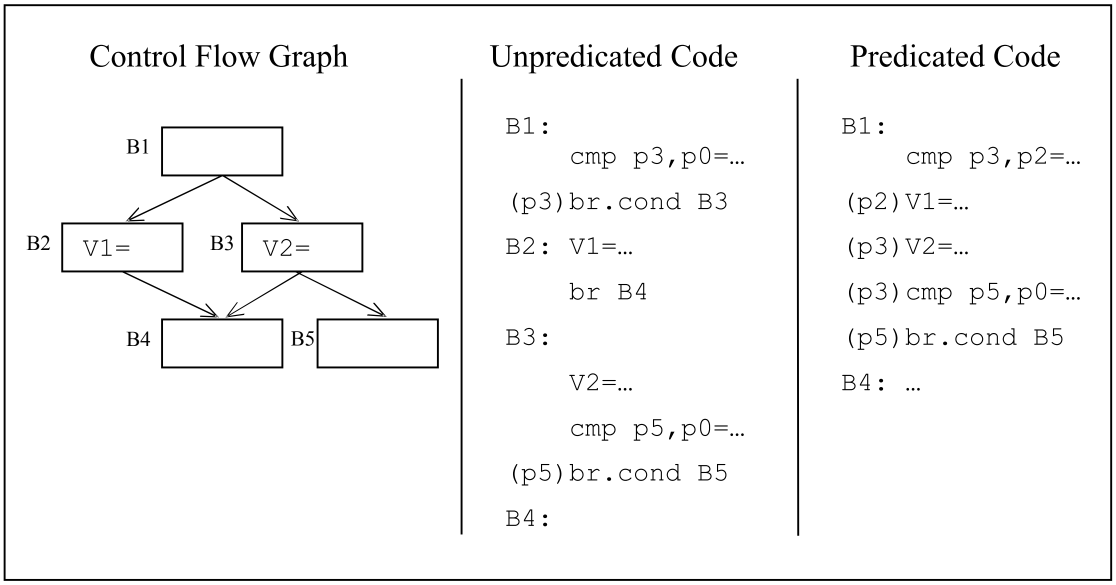
```
B1:
    cmp p3,p0=…
(p3)br.cond B3
B2: V1=…
    br B4
B3: V2= …
    …
B4: …
```

Predicated Code

```
B1:
    cmp p3,p2=…
(p2)V1=…
(p3)V2=…
B4: …
```

Figure 2 Unpredicated and If-converted Hammock

The notion that if-conversion eliminates branch mis-predictions is correct for "closed" regions like hammocks, where all branches can be eliminated. Theoretically it is possible that if-conversion transfers, but does not eliminate a branch mis-prediction. This scenario could happen if a branch remains in the if-converted region. Figure 3 is making the case. The predicated region again is a hammock with merge point B4, but it has an "exit" branch to a block B5 outside the region. In this case a conditional branch (instruction 5 in the predicated code) remains in the region. It could happen that in the unpredicated code the first conditional branch was mispredicted (instruction 2), but after if-conversion the remaining conditional branch is mis-predicted, so if-conversion is not guaranteed to be effective in this case. In practice we have not observed this scenario, but it seems the notion that if-conversion eliminates branch mis-predicts is valid for the case of "closed" regions only, where if-conversion can eliminate all branches.



Unpredicated Code

```
B1:
    cmp p3,p0=…
(p3)br.cond B3
B2: V1=…
    br B4
B3:
    V2=…
    cmp p5,p0=…
(p5)br.cond B5
B4:
```

Predicated Code

```
B1:
    cmp p3,p2=…
(p2)V1=…
(p3)V2=…
(p3)cmp p5,p0=…
(p5)br.cond B5
B4: …
```

Figure 3 If-converted Region with Exit Branch

Like any compiler optimization, if-conversion must consider trade-offs: cost of if-conversion includes code size increase (both static and dynamic) and execution time increase for execution paths in the if-converted region. The potential benefits, which include elimination of branch mis-predicts and potential code size decrease, must outweigh the costs. If-conversion also competes with other optimizations like speculation for resources, e.g. instruction slots. It is conceivable that if-conversion may enable or disable control-or data speculation by consuming hardware resources. From these considerations it is clear that the heuristics that govern if-conversion cannot be simple. In general the compiler has no perfect knowledge of branch mis-predicts, execution counts or the dynamic interaction of optimizations. The Intel compiler employs an if-conversion Oracle that carefully evaluates the benefits of if-conversion for a given region, and decides to if-convert only when the Oracle suggests the estimated average execution time for the predicated code is better than for the original (unpredicated) code.

Another benefit of if-conversion is that it enables the compiler to remove control flow in innermost loops for kernel-only software pipelining (Rau [65]). The problem of register allocation for predicated code will be discussed in chapter 6.

2.3 Architected Registers

The Itanium processor has a large number of architected registers supporting predicated instructions, control- and data speculation, register rotation and a dynamic register stack. Relevant for register allocation are the 128 general (integer) registers r0-r127, 128 floating point register f0-f127, 64 predicate registers p0-p63 and 8 branch registers b0-b7 (Figure 4). The register types are partitioned into preserved and scratch registers by the ABI [44]. The content of a preserved (=callee saved) register may not be destroyed by a callee (=a function called). If the callee is using a preserved register, it must restore the original value before return. The content of a scratch (=caller saved) register can be destroyed by a callee. Itanium has four registers representing constants: integer value zero in r0, floating-point value 0.0 in f0, floating-point value 1.0 in f1 and predicate value 1 in p0. Special integer registers are reserved for data access (r1 is the global pointer to access global data, r12 is the stack pointer and r13 the thread pointer) and the return address (b0). The floating point and predicate register files contain rotating registers f32-f127 and p16-p63 respectively. Unique for a processor are the 96 stacked integer registers, r32-r127, which are controlled by a special processor unit, the Register Stack Engine (RSE). Rotating registers and the register stack necessitate the distinction between architectural and physical registers. For example, architectural register r32 can be any physical register from r32 up to the number of stacked registers implemented by the micro-architecture. Note that some but not all stacked registers may rotate. The actual number of rotating stacked registers is specified by the alloc instruction. Sections 2.4 and 2.5 cover the details about stacked and rotating registers.

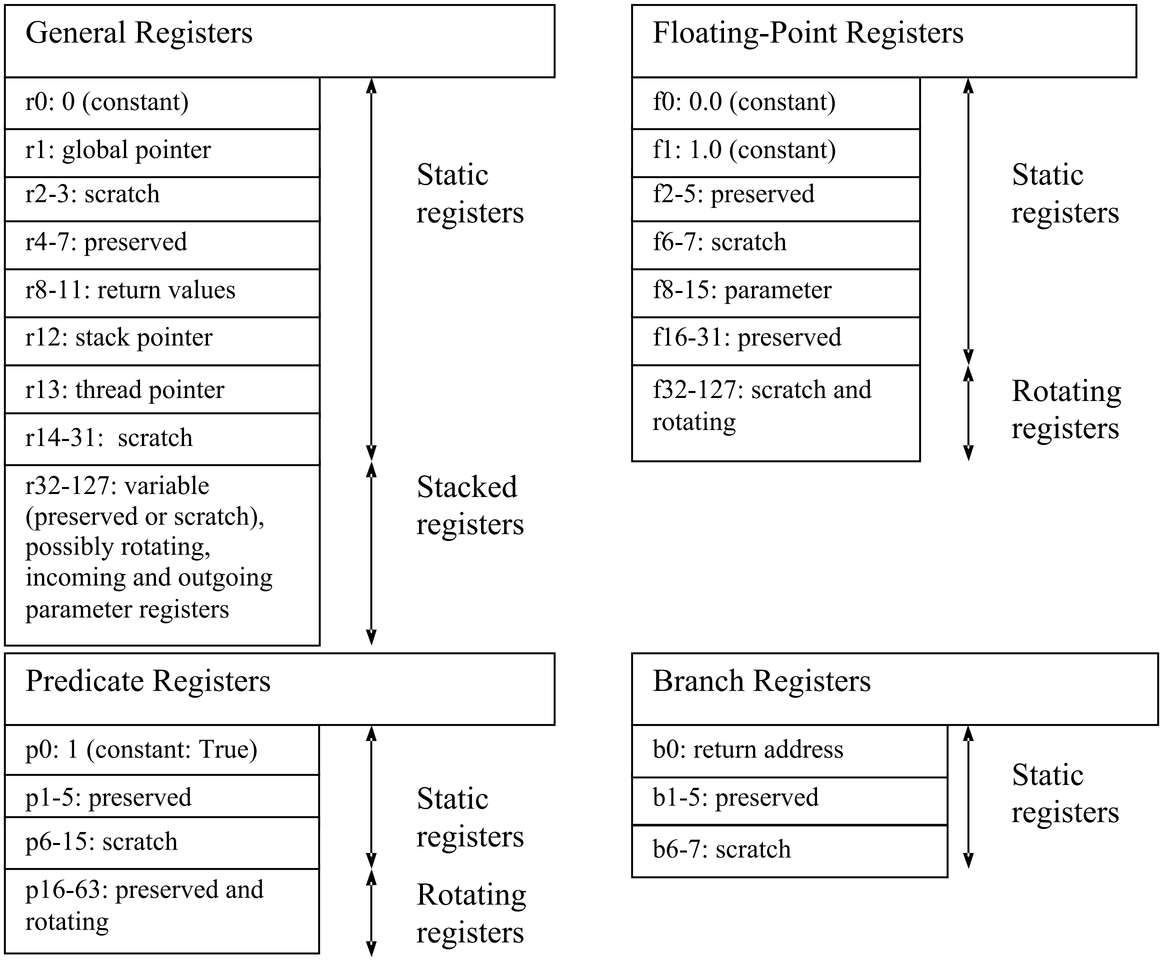


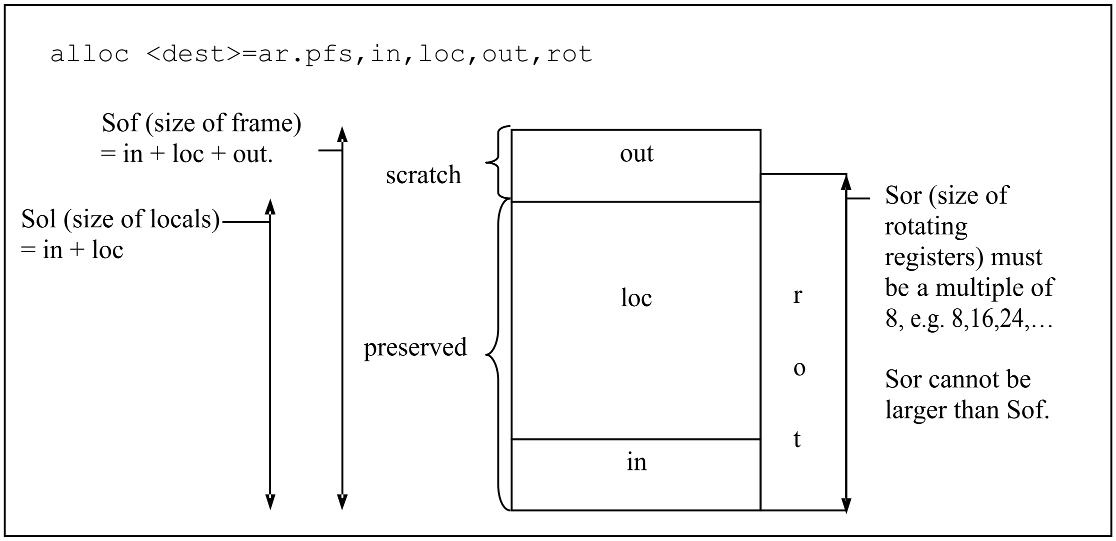
Figure 4 Register Files and Partitions

2.4 Register Stack Frame

On Itanium, each procedure has its own variable size register stack (Figure 5) with its own variable number of rotating registers. Any stacked register can be rotating, but the number of rotating registers must fit within the register stack and be a multiple of 8 (0, 8, 16 and so on registers can be rotating), starting at r32. At the bottom of the register stack (starting at r32) are up to 8 incoming argument registers. At the top of the stack are up to

8 outgoing parameter registers. The outgoing parameter registers are scratch and become the incoming arguments in the register stack of the callee. The number of incoming argument and outgoing parameter register is dictated by the Itanium ABI [44].

The *alloc* instruction (Figure 5), which is an example for an instruction that cannot be predicated, specifies the register stack frame of a procedure: the number of incoming parameters (in), the number of local registers (loc) and the number of outgoing parameters (out). The total number of registers in a register stack is in+loc+out <= 96. Usually a register stack has up to 8 incoming argument register and up to 8 outgoing parameter registers. Parameters that do not fit into the out section of the register stack are passed on the memory stack following the rules of the Itanium ABI [44]. The local registers are determined by the register allocator. The number of rotating stacked registers must be specified in the last parameter *rot* of the alloc instruction in routines that contain software pipelined loops. The architectural register ar.pfs contains fields that describe the register stack of the caller ("previous function state") and is saved into the destination register <*dest*> of the alloc instruction for register stack unwinding.

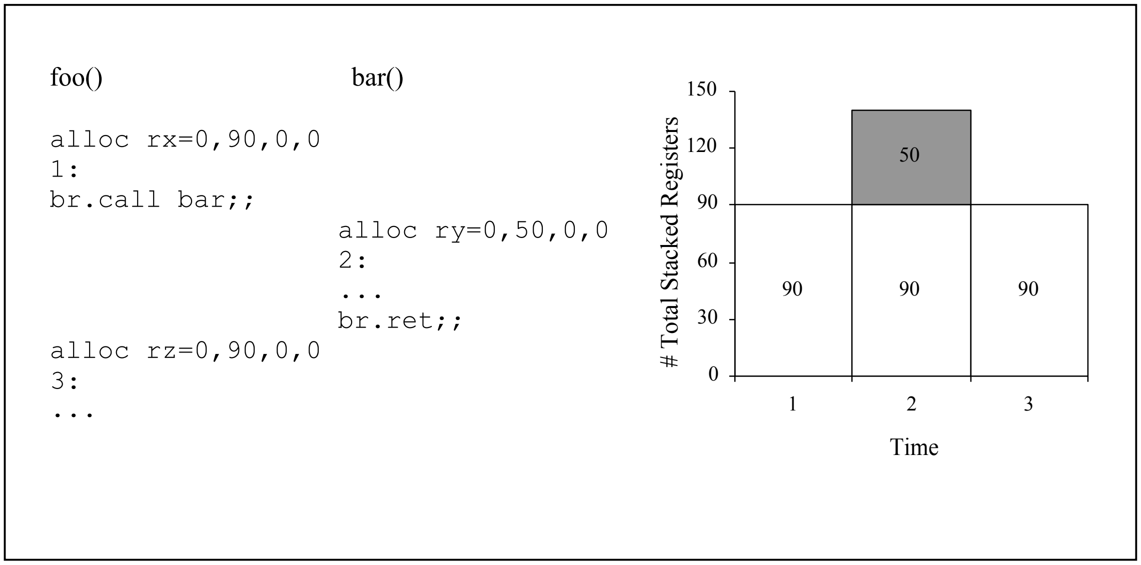


```
alloc <dest>=ar.pfs,in,loc,out,rot
```

Sof (size of frame)
= in + loc + out.
Sol (size of locals)
= in + loc
scratch
preserved
out
loc
in
r
o
t
Sor (size of rotating registers) must be a multiple of 8, e.g. 8,16,24,…
Sor cannot be larger than Sof.

Figure 5 Alloc instruction and Register Stack Frame

Figure 6 shows assembly snippets of a function foo() calling a function bar() and snapshots of the register stacks at 3 points in time: after the allocation of 90 stacked registers in foo (1 :), after the additional allocation of 50 stacked registers in bar (2 :) and after the return from bar (3 :). Note that to simplify matters the alloc instructions in Figure 6 don't specify rotating registers since the fact that some stacked registers can be rotating is irrelevant in this context. Combined, foo() and bar() use 140 stacked registers. Since more than 96 stacked registers are used on the call stack, the processor recognizes a stack register overflow at the call of bar() and saves 50 registers from the register file to memory so that the register stack frame of bar() can reside in the register file. The memory that contains the saved registers is called the "backing store" and is managed by the operating system. Similar, since registers allocated by foo have been saved and overwritten by operations in the callee, at the return from bar() the processor would recognize a stacked register underflow and restore registers from the backing store to the register file. The saves and restores of stacked registers are transparent to the program and controlled by the RSE. More details about the backing store are in Section 2.5.



```
foo()                      bar()

alloc rx=0,90,0,0
1:
br.call bar;;
                           alloc ry=0,50,0,0
                           2:
                           ...
                           br.ret;;
alloc rz=0,90,0,0
3:
...
```

Figure 6 Snapshots of Stacked Register Usage

The values of the procedure frame parameters are maintained in the Current Frame Marker (CFM) field of the Current Function State (CFS) application register. When a call is executed, the content of the CFM is copied to Previous Frame Marker (PFM) field of the Previous Function State application register. The caller's output area becomes the callee's register stack frame. The size of the local area is zero, and the initial size of the frame, which at this point consists of the input area only, is the size of the caller's output area. The stacked registers are renamed such that r32 becomes the first register on the stack. The alloc instruction creates the register stack frame for the callee. The input section of the local area in the new frame matches the output area of the caller's frame. In other words, the input registers in the callee's frame are the renamed registers of the caller's output area. This effectively passes the caller's register parameters to the callee. When the return executes, the CFM field is restored from the PFM field and the original register stack frame of the caller is re-instantiated. Figure 7 shows an example with a register stack frame of size 21 with 7 outgoing registers (r46-r52). After the execution of the call the register stack frame of the callee exists of 7 incoming registers. Registers r46-r52 have been renamed to r32-r38. The new register stack frame has been recorded in the sol ("size of local") and sof ("size of frame") fields of the CFM. There sol is 0, while sof is 7.

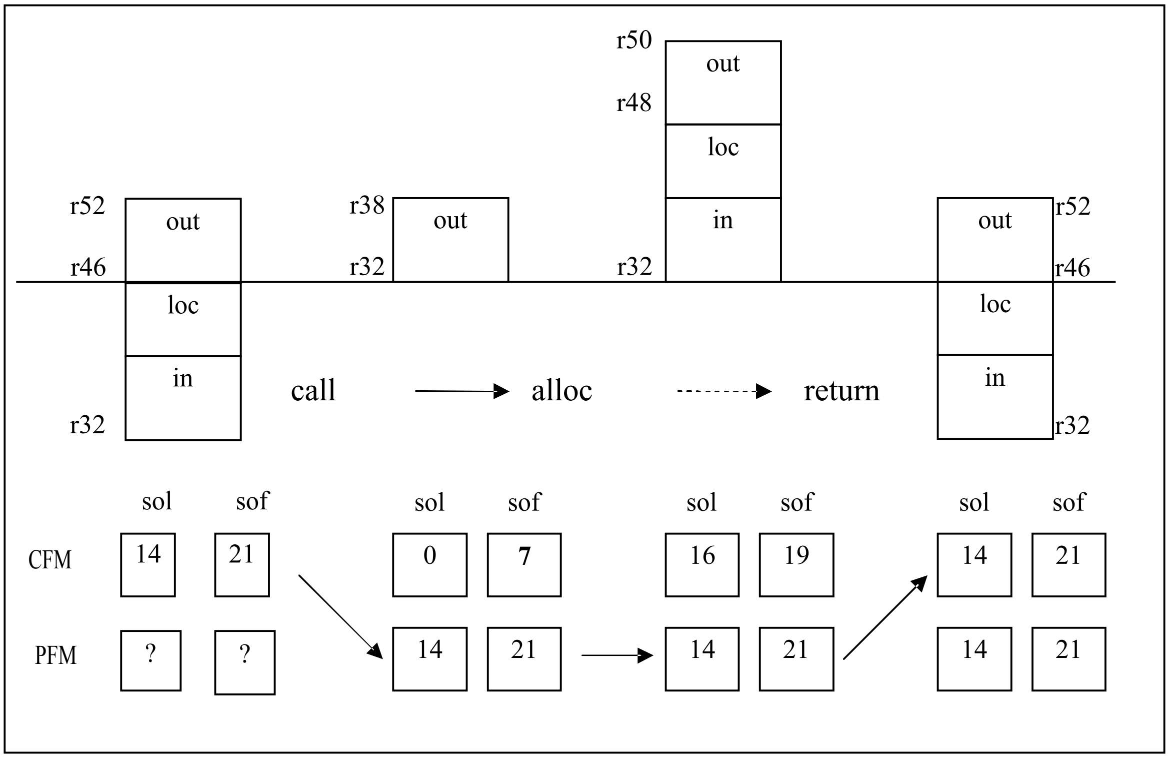


Figure 7 Register Stack - Growth and Shrinkage

2.5 Register Spilling and Backing Store

The *register stack engine (RSE)* manages the 96 stacked register partition of the physical register file as a circular buffer. The stacked partition of the register file is partitioned into three parts: mapped registers belong to some stack frame of a procedure on the call stack, unmapped registers do not belong to any frame and active registers (which can be viewed as special mapped registers) are the frame of the running ("active") procedure. Overflows occur when a new frame is allocated and overlaps with mapped registers. In this case, the RSE makes room for the new frame by spilling overlapped mapped register to memory, the *backing store.* Each process has its own backing store. Underflows occur at procedure returns when unmapped registers of the caller must be filled from the backing store. The RSE manages the register file and the backing store, with a set of internal pointers. Figure 8 shows a snapshot with three frames in the register file.

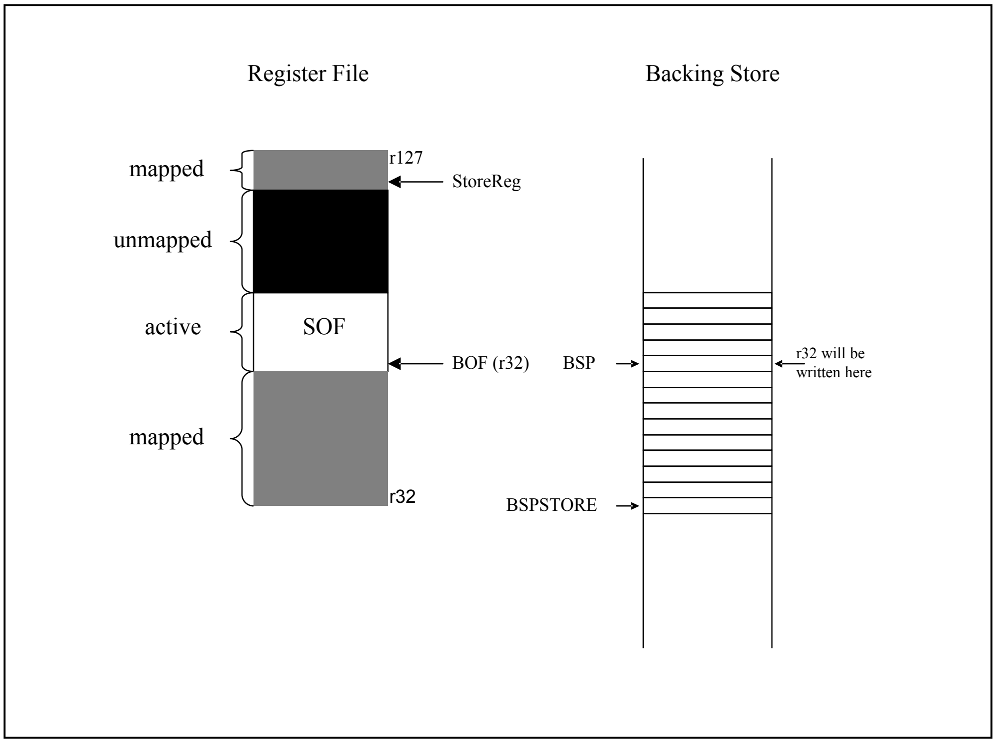


Figure 8 Register File, Frames and Backing Store

In the figure above BOF ("bottom of frame") points to the r32 of the active frame. Should the active frame get saved, its r32 would be saved to the backing store address which is hold in the BSP ("backing store pointer"). *StoreReg* points to the mapped register that is saved to address BSPSTORE in case of an overflow. In case of an underflow registers get restored starting from address BSPSTORE-8. The actual RSE actions depend on whether the call stack, and thus the register stack, will grow or shrink in future. It is clear that RSE register saves and restores ("RSE traffic") for an application increases proportionally to the size of register frames and the depth of the call stacks.

2.6 Speculation

Speculation means early execution of an operation. At this early point it is unknown if the result of the operation is needed. Itanium supports two major explicit forms of speculation: control- and data speculation. Both types of speculation are non-exclusive and can coexist.

2.6.1 Control Speculation

Control speculation (*breaking the branch barrier*) is an optimization that hoists a chain of instructions starting at a load above one or more controlling branches. Instructions can be divided into two classes: speculative and non-speculative instructions. Speculative instructions, which defer exceptions, may execute prematurely. In general, all arithmetical instructions, which write result to general or floating-point register, are speculative. Loads are non-speculative instructions that raise exceptions if they occur and cannot be speculated. Therefore Itanium provides speculative loads (ld.s, ldf.s, ldfp.s) in addition not non-speculative loads (ld, ldf, ldfp). A failed speculative load (e.g. due to a page fault) causes a deferred exception token to be written into destination register of the speculated load. For general registers, the token is an extra bit (NaT) for each register. Thus the general registers are 65 bits wide. When the NaT ("Not a Thing") bit is set (value is 1), the register contains a deferred exception token. Otherwise the NaT bit is clear (value is 0). In floating-point registers the deferred exception token is set when it contains a special zero encoding, the NaTVal. The speculative loads are the only producers of deferred exceptions tokes, which propagate across the chain of speculated instructions: the destination register will inherit the deferred exception token if it is set in any source.

Since control speculation can yield invalid result, a validation mechanism must be provided. For this Itanium provides a chk.s instruction (for both, general and floating-point registers), which must execute at the point of the original load when the result is needed. If the source register in the chk.s contains a deferred exception token, execution branches to recovery code, which re-executes a non-speculative instance of the speculative load and all instructions in the dependence chain, therefore clearing the deferred exception token in the chain, and branches back to the bundle after the chk.s. Example 1 shows a simple example for control speculation of a single load and a dependent add instruction. In case of an exception, recovery code executes and computes the result. Program execution continues at the bundle after the chk.s.

```
      Original Code           Control Speculated Code

      ld8    V3=[V1];;        ld8    V3=[V1]
      cmp.eq V10,p0=V3,0      ld8.s  V8=[V7];;
                              cmp.eq V10,p0=V3,0
      add    V5=V4,V3         add    V5=V4,V3
                              add    V9=V9,V8
(V10) br cont;;         (V10) br     cont;;
      ...                     ...
      ld8   V8=[V7];;     r1: chk.s  V8,rec
      add   V9=V9,V8
cont: ...               cont: ...
      ...                     ...
                        rec:  ld8    V8=[V7]    //recovery
                              add    V9=V9,V8   //code
                              br     cont;;
```

Example 1 Control Speculation with Recovery Code

Speculation can offer many benefits: it can decrease critical path length, increase ILP and hide memory latency. This is balanced by potential cost: First, there is the opportunity cost of wasted resources when the result is not needed. Second, an exception results in dynamic code duplication, chk.s branch overhead and potential I-cache pollution from executing recovery code. Third, careless speculation can increase critical path length. Finally, like any code hoisting optimization, control speculation can increase register pressure.

The register allocator must be aware of deferred exception tokens. For integer registers, st8.spill/ld8.fill instructions are defined to save/restore a general register and its NaT bit. The NaT bit of the spill/fill is stored in/restored from a preserved 64bit application register, the ar.unat. Bits 8:3 (six bits) of the memory address of the spill/fill determine the specific ar.unat bit that correspond to the spill source/fill destination register. For floating-point registers, stf.spill/ldf.spill save/restore a register without raising an exception if the source/destination register contains a NaTVal.

2.6.2 Data Speculation

Data speculation (*breaking the store barrier*) is an optimization that hoists a chain of instructions starting at a load above one or more ambiguous stores. A load and store are ambiguous, when it is unknown at compile (assembly) time whether the load and store address overlap. Itanium provides advanced loads (ld.a, ldf.a, ldfp.a), an advanced load check (chk.a) and load check (ld.c, ldf.c, ldfp.c) instructions for data speculation. When a chain of instructions is speculated, the result must be checked with a chk.a, which is similar to chk.s. When the chain is only a single load, a load check instruction can be used. An advanced load records information about its physical destination register, memory address and data size in the Advanced Load Address Table (ALAT) [42]. When a subsequent store overlaps, the processor invalidates the corresponding ALAT entry to indicate the collision. A load check only reloads the correct value when it finds no valid ALAT entry. As with control speculation, a chk.a branches to recovery code and re-executes the speculated instruction chain (Example 8), when an address overlap has happened. For data speculation, the code generator has to handle two performance issues: ALAT conflicts and ALAT collisions. On the first Itanium processor (“Merced”) the ALAT is a 2-way set associative cache with 16 entries per set. The four least significant bits of the physical ld.a destination register form the set index. When two ALAT live ranges interfere, the register allocator has to assign them two (mod 16) different physical registers to avoid ALAT conflicts. The register allocator can guarantee this only if both registers are in the same class: static, stacked or rotating. When the two registers are in different register classes, e.g. one static, one stacked, in general the compiler cannot derive their physical register number. On the Itanium-2 the ALAT has 32 entries and is fully associative [42] and the register allocator does not have to worry about collisions.

```
            Original Code             Data Speculated Code

                                      ld8.a  V4=[V1]
                                      add    V5=V4,V6
                                      ...
            st4   [V10]=V11           st4    [V10]=V11
            ld8    V4=[V1]            chk.a  V4, rec
            add   V5=V4,V6     cont:  ...

                               rec:   ld8    V4=[V1]    // recovery
                                      add    V5=V4,V6   // code
                                      br     cont       //
```

Example 2 Data Speculation with Recovery Code

2.6.3 Combined Control- and Data Speculation

Control- and data speculation can co-exist and be performed simultaneously to break both, branch and store barriers. Itanium supports control-speculative variants of advanced loads (ld.sa, ldf.sa, ldfp.sa). When such loads generate a deferred exception token, no ALAT entry for the (physical) destination registers will exist. Thus an advanced load check or a load check instruction validates the control-and data speculated result like for “pure” data-speculated code.

3 Review of Graph-Coloring based Register Allocation

This chapter gives an overview of the rich body of literature on graph-coloring based register allocation. While it cannot be complete, it does review many key ideas of the subject. For a perspective, a fundamental building block of Chaitin's *graph-coloring based register allocator* ("coloring allocator"), the simplification algorithm, was described by Kempe [47] in 1879.

3.1 Foundations

Register allocation solves the decision problem which symbolic register should reside in a machine register. A symbolic register represents a user variable or a temporary in a compiler-internal program representation. *Register assignment* solves the decision problem which specific machine registers to assign a given allocated symbolic register. Solutions to both problems must take into account constraints between symbolic registers. A coloring allocator abstracts the allocation problem to coloring an undirected interference graph with K colors, which represent K machine registers. The red thread of the relevant literature starts with Chaitin's paper [16], which describes the first complete implementation of a coloring allocator in an experimental IBM PL/I compiler. In a follow-up paper Chaitin describes - in "broad brush strokes" - the fundamental building blocks of coloring allocators [17]. Chow introduced priority-based graph coloring as part of the optimizing UCODE compiler [20]. Chaitin's and Chow's papers inspired many developments in the field. A short account of the history of graph coloring methods in computer science before Chaitin can be found in Briggs' thesis [12].

3.1.1 Chaitin-style Register Allocation

A Chaitin-style graph-coloring algorithm has six phases (Figure 9): "rename", "build", "coalesce", "simplify", "spill" and "select". At the start of the algorithm each symbolic register corresponds to a single register candidate node ("renaming"). This phase may split disjoint definition-use chains of a single variable into multiple disjoint live ranges. It also ensures contiguous numbering of candidates reducing memory requirements for dataflow-analysis and interference graph. Node interference relies on dataflow analysis to determine the live range of a node. The live range of a node consists of all program points where the register candidate is both live and available. Dataflow analysis is necessary only once, not at each build step. The “build” phase constructs the interference graph. The nodes in the interference graph represent register candidates. Two nodes are connected by an interference edge when they cannot be assigned the same register. The number of edges incident with a node are the degree of the node. Building the interference graph is a two pass algorithm. In the first pass, starting with the live out information, node interference is determined by a backward sweep over the instructions in each basic block. Interference is a symmetric relation stored in a triangular matrix. This is usually a large, sparse bit matrix inadequate for querying the neighbors of a given node. To remedy this for each node an adjacency vector is allocated in a second pass. The length of the vector is the degree of the node. It contains all neighbors of the node.

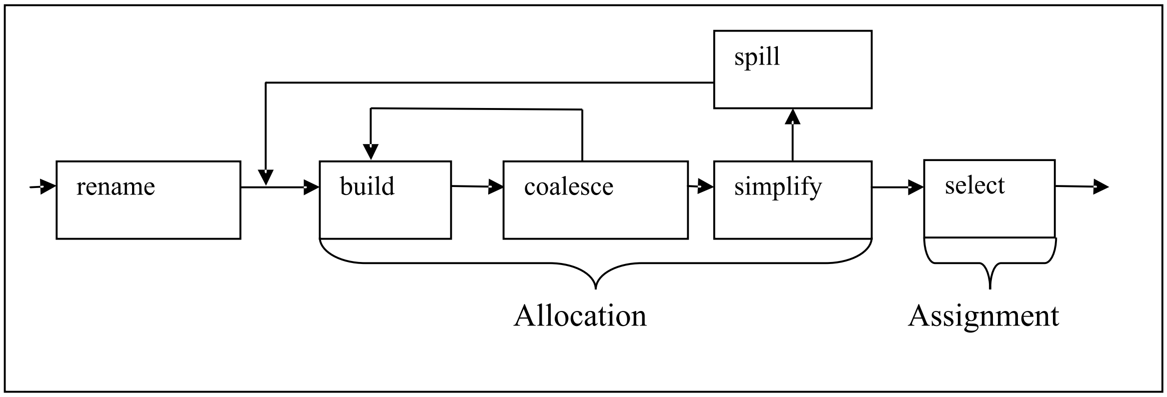


Figure 9 Chaitin-style Register Allocator

The next phase, “coalescing” (aka “subsumption”, “node fusion”), is an optimization not needed for solving the register allocation problem, but was introduced in the original Chaitin allocator. It fuses the source and destination node of a move instruction when the nodes do not interfere. This reduces the size of the interference graph and eliminates the move instruction, since source and destination get assigned the same register. Chaitin’s original implementation can coalesce any possible pair of nodes. This form of coalescing is called “aggressive coalescing”. After possibly several iterations of “coalesce” and (re-) “build”, the simplification phase iterates over the nodes in the interference graph using simple graph theory to find candidates that can be allocated to machine registers: when a register candidate has fewer than K interference edges (*low degree* node that has fewer than K neighbors), then it can always be assigned a color. Low degree nodes and their edges are removed from the graph ("simplify") and pushed on a stack ("coloring stack"). Node removal may produce new low degree nodes. When only high degree ("significant") nodes that have K or more neighbors are left, then simplification is in a blocking state. It transitions out of a blocking state using a heuristic-based priority function that determines the "best node" to be removed from the graph. A node that is removed from the graph in blocking state is "spilled" and appended to a spill list. Spilling is an allocation decision and a spilled node will reside in memory (stack) rather than in a register. The edges of a spilled node are removed from the graph, so new low degree nodes can get exposed and simplification continues until all nodes have been pushed to the coloring stack or appended to the spill list. The cost function that decides on the "best node" estimates the execution time increase caused by spill code normalized by the degree of the node. The higher the degree the less likely a node will be allocated a register. The formula for the cost function is in Equation 1. The sum is over all basic blocks that contain a reference to node n. In Equation 1 d_i is the number of definitions of n, u_i the number of uses and f_i an estimate of the execution frequency of basic block i. The expected execution time of a load and store on specific target architecture is S and L respectively.

$$select_{Chaitin}(n) = Cost_{Chaitin}(n) = \frac{Cost(n)}{Degree(n)} = \frac{\sum_{i} (S \times d_i + L \times u_i) \times f_i}{Degree(n)}$$

Equation 1 Cost Function in Chaitin Allocator

The node with the smallest cost is picked for spilling. When the spill list is not empty at the end of "simplify", the "spill" phase substitutes a spilled candidate with possibly multiple new register candidates. In the worst case a new candidate is introduced for each definition and use of the original register candidate. Figure 10 shows the original code and the code a spilling. The register candidate of a definition is replaced with a new register candidate rc1, which is spilled immediately after definition to the address contained in the spill register. Similar, a fill is inserted before a use. Note that the move of the spill address into register sp is not shown in Figure 10. The replacement of the original spill candidate rc with new candidates rc1 and rc2 splits the original live range for rc into two small live ranges for rc1 and rc2. When spilling has occurred, the allocator must restart at the build phase and iterates until all register candidates are colored.

```
        Original Code                     After Spilling

1:      add   rc=…                        add rc1=…
2:                                        st8[sp]=rc1
3:      …                                 …
4:                                        ld8 rc2=[sp]
5:      sub   …=rc                        sub …=rc2
```

Figure 10 Illustration of Spill Code Insertion

After a few build-coalesce-simplify cycles, the spill list is empty. At this point the allocation problem is solved: original candidates that are in the coloring stack have been allocated to registers. The coloring stack is fed into the coloring or “select” phase. This phase picks one node at a time starting from the top of the stack and assigns a color to the node while it ensures that interfering nodes receive different colors. For this the adjacency vectors are used: colors that have been assigned to neighbors get blocked and cannot be assigned to the current node.

The order in which candidates are assigned registers impacts allocation. Simple examples show that some graphs with nodes of degree K or higher can be colored with K or less colors. Briggs et al. [13] used this insight and modified the Chaitin allocator by pushing all nodes onto the coloring stack during simplification. The actual spill decision is delayed until after register assignment. This delay technique is known as *optimistic coloring*, since significant nodes could get assigned a register rather than being spilled like in Chaitin’s original allocator (Briggs [12]). With optimistic coloring (Figure 11), register assignment solves the allocation problem: when a candidate has been assigned a register, it has been allocated to a register. When all candidates on the coloring stack get assigned a register, the algorithm terminates. Otherwise it spills the unassigned candidates and restarts allocation at the build phase.

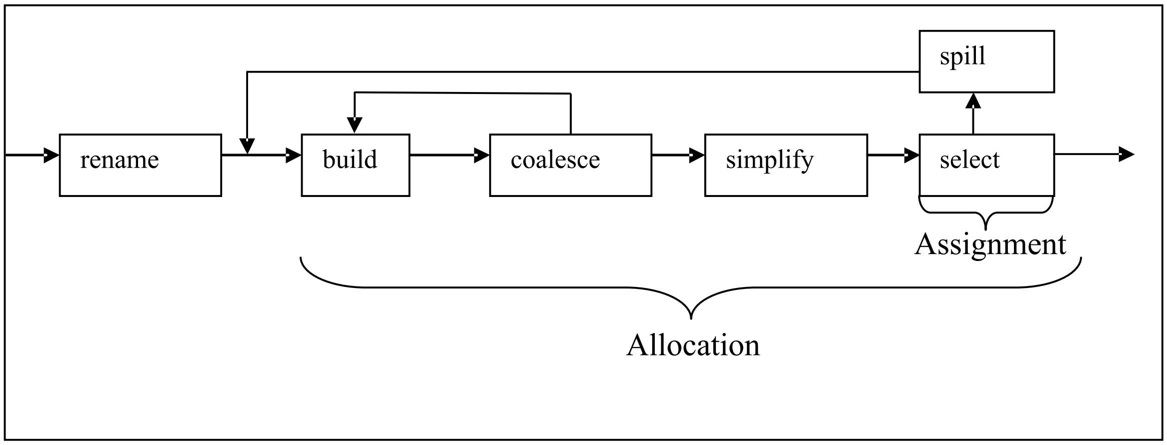


Figure 11 Chaitin-style Allocator with Optimistic Coloring

3.1.2 Priority-based Register Allocation

Priority-based register allocation was introduced by Chow [20]. This coloring allocator uses the basic building blocks of a Chaitin-style allocator except for the coalescing phase. A *priority function* estimates the execution time decrease when a live range is assigned a register rather than residing in memory. The live ranges are composed of one or more live units. A live unit is a basic block where a given symbolic register could reside in a register. The more expensive (relative to Chaitin) representation of a live range supports *live range splitting*, which splits a given live range into a set of smaller live ranges when no color can be assigned. Rather than spilling the entire live range, live range splitting dissolves a given live range into new candidates that could become colorable. The phases of a Chow allocator (Figure 12) are “rename”, “build”, “simplify”, “select I”, “split” and “select II”. A Chow allocator works on two pools of nodes: the constrained and the unconstrained node pool. These pools are the result of the Chow simplification phase: unconstrained live ranges, which are the low degree nodes with less than K neighbors, are collected in the unconstrained pool, and constrained live ranges, which are the significant nodes with K or more neighbors, are collected in the constrained pool.

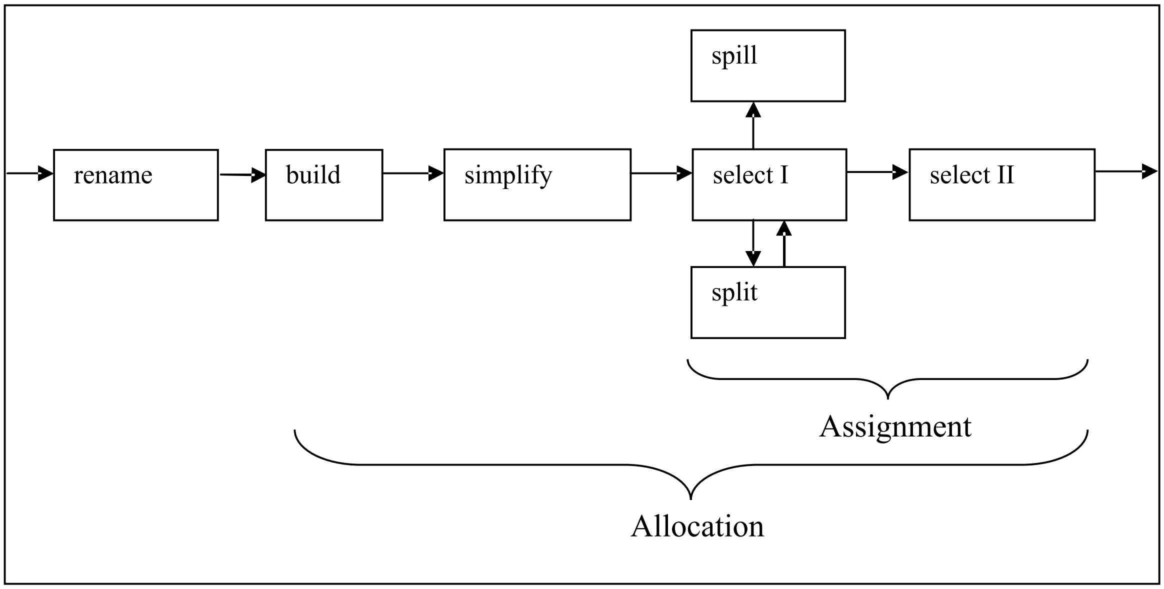


Figure 12 Chow-style Allocator

Unlike in a Chaitin allocator, the simplification phase sweeps over the nodes only once. In assignment phase “select I” a priority function estimates the potential execution time savings from a register assignment. This function is normalized to the length of the live range. The rational is that longer live ranges get lower preference since they consume registers longer reducing the chances of other live ranges to be assigned registers. The live range with the highest priority is assigned a register first. Equation 2 shows the priority function of a live range n: the sum is over all basic blocks that contain a reference to node n. d_i is the number of definitions, u_i the number of uses m the number of reconciliation moves, which could be necessary to reconcile assignments to split live ranges segments, and f_i an estimate of the execution frequency of the basic block i. The expected execution time of a load, store and move is L, S and M respectively.

$$\boxed{select_{Chow}(n) = priority_{Chow}(n) = \frac{Benefit(n)}{Length(n)} = \frac{\sum_i (S \times d_i + L \times u_i - M \times m_i) \times f_i}{Length(n)}}$$

Equation 2 Priority Function in a Chow Allocator

The numerator in the priority function is similar to the numerator in Chaitin's cost function, except that it is interpreted as the execution time benefit of register assignment and models the reconciliation cost ($M \times m_i$) for a split live range.

When no color is available in the priority select phase, the Chow allocator is in a blocking state. It transitions out of a blocking state by splitting a live range into two or more live range segments, which form new live ranges. Splitting starts at the first block where the node could be assigned a register and determines in a breadth-first search the maximal segment (= first new live range) that can be colored. This is done recursively until the original node is split entirely into smaller live ranges and the constrained and unconstrained pools are updated accordingly. When all live units in a live range are constrained, the node cannot be split. In this case the Chow allocator spills the node by removing it from the constrained pool ("spill"). After coloring of the constrained nodes has terminated, all unconstrained nodes are colored in "select II". The original Chow allocator does not support coalescing, operates at a high-level intermediate representation, assumes all candidates reside in memory initially, and reserves machine registers for spilling, using fewer colors than machine registers available. The complexity of the select I phase is $O(K \times (L - K))$, where L is the total number of live ranges (Chow [20],[22]).

3.2 Worst-case Time and Space Analysis

This section reviews time and space complexity of the phases of a Chaitin-style allocator. In its simplest form "rename" is a linear pass over the control flow graph renaming symbolic registers. It may allocate a table to record information about each candidate. The impact of renaming on the intermediate representation is illustrated in Figure 13. Renaming can compact the candidates since the compiler may introduce new symbolic registers and dispose used symbolic registers before it calls the allocator. Compaction reduces memory requirements in particular for the interference graph. Rename may also select candidates: for example, the Intel compiler, which is discussed in more detail in chapter 4 , separates allocation for integer and floating-point candidates. In this case the "rename" phase will select only the candidates of the class that gets allocated. A more elaborate implementation of "rename" could also split a live range into disjoint components. For example, the live range of D in Figure 13 has two disjoint components from lines 1-5 and lines 10-18 respectively and could get splits into two candidates RC3 and RC4.

```
       Source          Intermediate Representation        after Rename

 1:    D=A+B           add V5=V4,V3                       add RC3=RC2,RC1
…      …               …                                  …
 5:          =D              =V5                                 =RC3
…      …               …                                  …
10:    D=              V5=                                RC4=
…      …               …                                  …
18:          =D              =V5                                 =RC4
```

Figure 13 Illustration of simple "rename" phase

Live ranges formation as part of the "build" phase relies on available variable and live variable analysis. The live range of a variable is the set of all program points where the variable is both live and available. Time and space complexity is similar for both standard bit vector-based dataflow analysis algorithms (Figure 14), except that available variable analysis does not require the *kill* vector. The notation in Figure 14 follows Aho et al. [1]. It assumes that a control flow graph is normalized with two distinguished nodes: a single START and a single EXIT node. For a reducible control flow graph the deepest loop nesting level of the function is an upper limit for the trip count of the dataflow solver. When the control flow graph is irreducible, worst-case time for a dataflow routine can be quadratic in the number of basic blocks.

Available variable analysis is used to identify undefined variables in acyclic code and to stop live range extension for non-strict live ranges at basic block boundaries where the live range is undefined. A live range is non-strict when there is a path from the function entry to a use that may contain no definition. When a non-strict live range is contained in a loop, its live range spans the entire loop. Figure 15 has an example for a non-strict live

	Live Variable Analysis	Available Variable Analysis
Lattice	Set of Variables	Set of Variables
Top	T (= Empty Set)	T (= Empty Set)
Meet	$\cup$ (= Set Union)	$\cup$ (= Set Union)
Boundary	$IN\left[EXIT\right] = \mathrm{T}$	$OUT\left[START\right] = \mathrm{T}$
Initialization	$IN\left[B\right] = \mathrm{T}$ for each basic block B	$OUT\left[B\right] = \mathrm{T}$ for each basic block B
Transfer	$F_B(X) = Gen(B) \cup (X \setminus Kill(B))$	$F_B(X) = Gen(B) \cup X$
Equations	$IN(B) = F_B(OUT(B))$ $OUT(B) = \bigcup_{S \in Succ(B)} IN(S)$	$OUT(B) = F_B(IN(B))$ $IN(B) = \bigcup_{P \in \Pr ed(B)} OUT(P)$
Direction	Backward	Forward

Figure 14 Live Variable and Available Variable Analyses

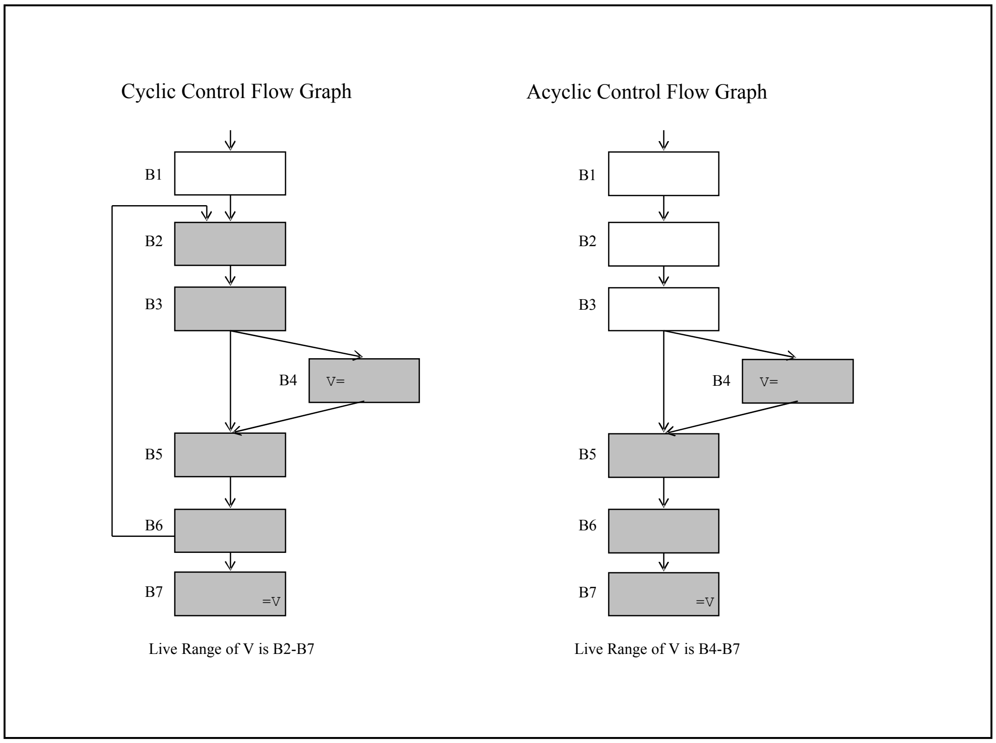
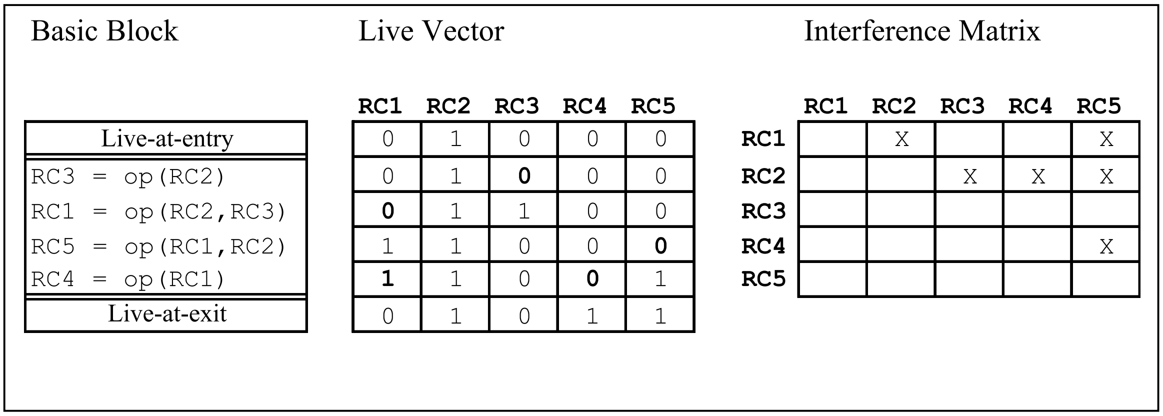


Figure 15 Example for a non-strict Live Range

range V. In the cyclic case, the live range of V extends from the exit of pre-header B1 to the use of V. In the acyclic case, the live range extends from the definition to the use V.

The result of live range construction is live vectors at the exit of basic blocks. To construct the interference graph, the allocator sweeps backwards over the instructions of the basic block updating the live vector at each instruction. In parallel it records interferences in the triangular interference matrix. Figure 16 shows a snippet of a basic block with five register candidates. Candidates RC2, RC4 and RC5 are live at exit. In the last instruction RC 4 is defined. Thus it interferes with all candidates live and the interferences with RC2 and RC5 are recorded in the interference matrix. Since RC4 is defined, RC4 is then removed from the live vector. Also, in the last instruction RC1 is used. Thus it is recorded as live in the live vector. There is no interference at this step between RC1 and RC4. The interference matrix shows the interferences after all instructions in the block have been visited. After the interference matrix has been recorded, the degree for each variable is known. In the second pass of the "build" phase all neighbors of a node are recorded in the adjacency vector. In the worst case the adjacency vectors can consume even more space than the interference matrix. This can happen for example for a complete graph with |V| nodes. A graph is complete when each pair of nodes is connected by an edge. In a complete graph with |V| nodes each node n has degree (n) =|V| - 1.



Basic Block

Live-at-entry
RC3 = op(RC2)
RC1 = op(RC2,RC3)
RC5 = op(RC1,RC2)
RC4 = op(RC1)
Live-at-exit

Live Vector

RC1	RC2	RC3	RC4	RC5
0	1	0	0	0
0	1	**0**	0	0
0	1	1	0	0
1	1	0	0	**0**
1	1	0	**0**	1
0	1	0	1	1

Interference Matrix

	RC1	RC2	RC3	RC4	RC5
RC1		X			X
RC2			X	X	X
RC3					
RC4					X
RC5					

Figure 16 Interference Graph Construction Scheme

Figure 17 shows the interference graph for the basic block snippet in Figure 16. Assuming K = 2, the simplification phase will push node 3, which has only one neighbor, on the coloring stack. Since 1 < K=2, this node is "unconstrained" and guaranteed to get assigned a register. After removing node 3, simplification is in a blocking state. Based on the cost function, it may pick node 4 as a spill node and place it "optimistically" onto the coloring stack, as illustrated in Figure 17, or in a spill list. The remaining nodes, 1, 2 and 5 still form a clique of 3 nodes, thus simplification is still blocked. It may decide to remove node 1. At this point node 2 and 5 have fewer than 2 neighbors and are pushed onto the coloring stack, terminating simplification. The select phase will assign colors to the candidates in reverse order they have been pushed onto the stack.

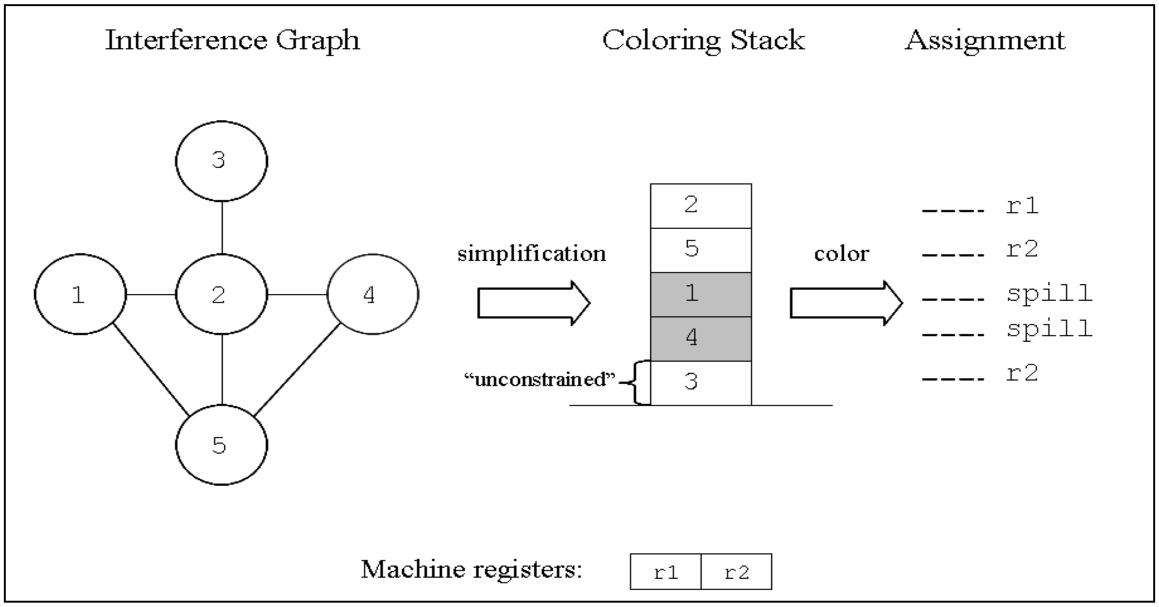


Figure 17 Simplification Phase and Coloring of Interference Graph

Figure 17 shows one possible assignment. Since node 1 and 4 did not get a register assigned, they get spilled and allocation restarts. In this particular example original Chaitin and optimistic coloring would produce identical spill code. In this example, the influence of node order on the allocation and assignment result is visible ("NP-noise", cf. Briggs [13]): assuming that node 2 is the cheapest node to spill in the first blocking state of the simplification phase it will be removed from the interference graph together with its neighbors. Then the remaining nodes will have only one neighbor and become unconstrained. In this scenario only node 2 may get spilled. So the allocation outcome depends on the assignment and the order of the nodes 1, 4, and 5 in the final coloring stack. If nodes 1 and 4 end up on the top of the stack and are assigned r1 and r2 respectively, node 5 will be spilled. But if nodes 1 and 4 are assigned r1, only node 2 must be spilled. The simplification algorithm can be implemented in time complexity $O(N \times \log(N))$ where N is the number of nodes. The assignment (or coloring) phase is a single linear pass over all nodes on the coloring stack.

3.3 Developments

Since Chaitin's work literature on register allocation progressed proposing new extensions, heuristics, scopes and alternatives. These developments can be summarized in six categories (Table 7).

Table 7 Research Categories and Goals

	Category	Goals
1	Spill code reduction	Performance (heuristics, remat., premat.)
2	Scoping	Compile time, Performance
3	Extensions	Functionality (e.g. load pairs), Performance
4	Coalescing	Performance, Compile time
5	Alternative Models	Compile time, Performance
6	Theory	Complexity analysis, Polynomial time solutions

First, spill code reduction improves on Chaitin's spilling heuristics to reduce spill and fill instructions. Splitting methods are heuristics to split live ranges into smaller pieces. Smaller live ranges should have less interference and could yield better allocation results. Second, the goal of scoping is to improve register allocation compile time or extend allocation scope, for example, allocating candidates across procedure boundaries rather than within a procedure. One idea to save compile time is to partition the control flow graph into a disjoint set of regions, perform register allocation per region and reconcile allocations at region boundaries. Implicitly, the walk over the regions also prioritizes register candidates and can impact allocation and thus run-time performance. Third, over time researchers and practitioners have implemented and proposed extensions to the classical coloring allocator to cope with architecture peculiarities. Fourth, coalescing almost is a field of its own, separate from register allocation. Fifth, in addition to graph-coloring based register allocation many other approaches have been proposed. Finally, there is rich amount of literature on the theory of register allocation. The remainder of the section surveys the six categories.

3.3.1 Spill Cost Reduction

The goal is to issue as little spill code as possible. To this end clever heuristics including preferencing, rematerialization, live range shrinking (a.k.a. pre-materialization) and live range splitting methods get employed.

3.3.1.1 Spill Heuristics

Two references of a live range are **close** when no other live range dies ("goes dead" in Chaitin [17]) between them. In other words, two references of a live range are close when no other live range ends between them. In this case, no new register can become available at the second reference. So, in the relevant case that the second reference is a use, the fill would have to use the same register the candidate is assigned to at the first reference and load the same value. Effectively the fill becomes a dead instruction. Chaitin's spilling heuristics a) exploit this fact and attempts to avoid spilling in basic block when "closeness" for live range references can be detected, and b) replace loads with simpler operations when possible (Rule 1 below, "rematerialization"):

- Rule 1: If a value is easy to re-compute, do it.
- Rule 2: If two uses of a live range are close, don't reload at the second use ("load forwarding")
- Rule 3: Don't spill a live range when all its uses are close
- Rule 4: If a use is close to its definition, use the stored value directly and don't fill ("store forwarding")
- Rule 4': If the use is close to its definition, and both references are within one basic block, don't spill the live range

It can be more effective to re-compute the value of a live range rather than spilling it. Trivial examples are live ranges that represent constants or easy to re-compute values like stack pointer + offset. This technique is called "rematerialization". Chaitin [17] uses it as a spill heuristic. Briggs [12] generalizes Chaitin's observation and constructs a dataflow framework to model rematerializable live ranges. Figure 18 illustrates Chaitin’s rules for two live ranges X and Y in one basic block. When no rule is applied, X is filled before each use. With rule 1, the load is replaced by an assumingly cheaper add instruction. Rule 3 is a generalization of rule 2: When all uses are close, only the first load of X is necessary. In case of live range Y there is a definition and use in one basic block. When no rule is applied, Y is spilled after the definition and filled before the use. If use and definition are close, rule 4 applies and only the spill is needed. If Y is local live range and the basic block contains all its references, it does not get spilled at all based on rule 4’.

Spilled X	Rule 1	Rule 3	Spilled Y	Rule 4	Rule 4’
load X	add X=sp,12	load X	def Y	def Y	def Y
use X	use X	use X	store Y	store Y	…
…	…	…	…	…	
load X	add X=sp,12				
use X	use X	use X	load Y		
			use Y	use Y	use Y
load X	add X=sp,12				
use X	use X	use X			

Figure 18 Illustrations of Chaitin's Spill Rules

Bernstein et al.[7] introduces the “best of three” simplification heuristic, which decides which node to spill when the Chaitin allocator is in blocking state. Chaitin’s cost function (see Equation 1) prefers spilling a node with low cost and high degree, but it ignores register pressure. Bernstein et al. add weighted approximations to the cost function and use the inverse of the square of the degree of a node. Using the square of the degree rather than the degree makes it more likely that high degree nodes get spilled, which in turn could expose more unconstrained nodes ([7]). The three Bernstein spill heuristics are listed in Figure 19.

$$h_1(n) = \frac{Cost(n)}{Degree^2(n)} \qquad (1)$$

$$h_2(n) = \frac{Cost\ (n)}{Area\ (n) \times Degree\ (n)} \qquad (2)$$

$$h_3(n) = \frac{Cost(n)}{Area(n) \times Degree^2(n)} \qquad (3)$$

Where:

$$\text{Area(n)} = \sum_{I \in \text{LIVE(n)}} 5^{\text{depth(I)}} \times \text{width(I)}$$

LIVE(n): set of all instructions where n is live at exit
depth(I): loop nesting level of instruction I
width(I): # of simultaneous live register candidates at instruction I

Figure 19 Bernstein et al: Three heuristic Spill Functions

Bernstein et al. experiment with all three heuristic functions, where Area(n) is a measure for the weighted register pressure of live range n (Figure 19). In their experiments, none of the 3 heuristics is found to be superior to others, but the minimum of all three actually does give a superior select routine in all their tests:

$$select_{Bernstein}(n) = \min(h_1(n), h_2(n), h_3(n))$$

Equation 3 Modified Cost Function in a Chaitin Allocator

3.3.1.2 Splitting

Cooper and Simpson [24] exploit "containment" to reduce spill code: a live range L1 is contained in a live range L2, when L1 is **not** live at **any** definition or use of L2. The idea is that spilling a live range that is contained in another, is in-effective since it does not reduce interferences or provide any benefit for the spill cost. "Containment" is encoded as a directed interference graph: when L1 is contained in L2, there is a directed edge from L1 to L2. When neither L1 is contained in L2 nor L2 is contained in L1, and both live ranges interfere, the interference edge remains un-directed. Containment–based splitting is a lazy technique: for a spilled live range L it estimates the cost of splitting neighbors that contain L. If this is not effective the algorithm tries to split L across live ranges that are contained in L. For example, assume L2 is spilled and L2 is contained in L1. The algorithm spills L2 "around" L1 first if this results in faster code (as estimated by heuristics in the algorithm). Otherwise, it attempts to spill L1 "around" live ranges it contains. The containment-based algorithm would split L1 at the boundaries of live ranges that it contains, if it (likely) benefits run-time performance.

Bergner et al. [6] introduce interference region spilling. The interference region for two live ranges is the set of program points where both live ranges are simultaneously live. The idea is that spilling a live range in an interference region only ("partial spilling") is cheaper than spilling the entire live range. It is a lazy technique: for any spilled live range it evaluates the cost of only partially spilling the live range. For each spilled node their algorithm groups edges into K classes, one class for each color. Each class represents an interference region and contains interference edges to neighbors assigned color k. A color k is chosen that minimizes overall spill costs and the edges in class k are removed from the interference graph. When a live range L1 is contained in live range L2, the interference region for L1 and L2 is all of L1. In this case, interference region spilling and Chaitin spilling give the same results. The basic idea of splitting live ranges in "zones" of register pressure is also mentioned in Section 5 in Chow [22].

Pre-materialization shrinks re-computable live ranges *before* register allocation, when there is no cost to do so. Baev and al. [5] apply this technique in the HP-UX Itanium compiler, when empty slots are found where the "rematerializable" live range can be pre-computed without impacting execution time. This reduces register pressure but does not increase schedule length. A more aggressive implementation could trade off the register pressure reduction with schedule length increase.

Bernstein et al.[7] describe a splitting technique called "cleaning". It is applied at a basic block level in the first two iterations of a Chaitin allocator: when a live range is spilled, only one load (store) is inserted at the first use (definition), independent of the number of references to the live range in the block. Also, the live range is renamed and becomes local in the block. Intuitively "cleaning" should reduce spills in regions of low register pressure. “Cleaning” is aggressively applying Chaitin’s heuristics by ignoring “closeness”. Since this could result in more allocation iterations, the technique is limited to the first two iterations only.

3.3.1.3 Preferencing

Preferencing methods attempt to reduce spill code by influencing the register assignment. The direct method, which is usually applied in region-based allocators, attempts to assign the same register in all regions. Other methods are more indirect and opportunistic. They attempt to influence the ordering of the nodes on the coloring stack for both constrained and unconstrained live ranges so that the eventual assignment is more likely to yield a better allocation. For example, Lueh and Gross [54] compute the benefits for assigning a preserved or scratch register to a live range. The data is used to sort live ranges that contain function calls on the coloring stack prior to coloring. By examples they show that this can result in better allocations. Koseki et al. [50] developed an elaborate preferencing allocator. They introduce four classes of register preferences (dedicated, limited, preferred, and dependent, build a register preference graph (RPG), which is a weighted directed graph where each node represents a candidate, register, or register class, and each edge represents a dependence. The coloring stack is partially ordered based on simplification precedence. This gives a lattice for the candidates. In their selection phase they sieve candidates starting at the top elements in an attempt to satisfy the preference that gives the most benefit. But their method is compile time intense and performance gains are not certain.

3.3.2 Scoping

In this category methods address compile time and performance. The contributions can be broken down into four sub-categories:

- Extension of scope to interprocedural allocation
- Partitioning of control flow graph into allocation regions
- Partitioning of interference graph
- Partitioning of candidates

The sub-categories are not independent. For example, smaller allocation regions usually result in a smaller interference graph.

Interprocedural methods attempt to color candidates across call boundaries. Chow [21] describes a simple method where the routines in the call graph are visited in reverse-post order ("depth-first order"). The allocation results in the callee are used by the caller to assign registers, which are unused by the callee, to live ranges that cross that call. This can avoid spilling of scratch registers at call sites and spare preserved registers. Steenkiste and Hennessy's method [73] is similar to Chow's. They also traverse the call graph bottom-up. Cycles in the call graph are broken by replacing strongly connected components with single compound nodes. They don't necessarily follow strict calling conventions when it is beneficial to do so. Usually the interprocedural allocator runs out of registers in the routines on top of the call stack. If this happens, their allocator falls back to a regular (priority based) intra-procedural allocator.

Callahan and Koblenz [15] describe a general region allocation scheme ("Hierarchical Graph Coloring"). They partition the control flow graph into a set of tiles. Tiles are sets of basic blocks with additional properties, so that a tile tree can be constructed: two tiles are either disjoint or contained (tile 1 is subset of tile 2 or vice versa) and there is a single root tile. Then graph-coloring is applied to each tile (region) in a bottom-up walk of the tree. At tile boundaries the allocations are reconciled. Reconciliation is necessary since a live range L1 may get assigned register r1 in tile 1 and register r2 in tile 2. At the tile boundary of tile 1 and tile 2 a "reconciliation" move from e.g. r1 to r2 must be inserted. Hierarchical graph coloring covers loop trees and graph-partitions based on single-entry single-exit ("SESE") regions. It is noteworthy that their allocator uses "pseudo" registers, which are assigned physical registers in a reconciliation phase. Norris and Pollack [61] pursue region based allocation in a similar fashion, but based on the program dependence graph ("PDG"). Statements guarded by the same control statement form an allocation region. Their region may have multiple exits. Fusion-based allocation partitions the control flow graph into arbitrarily disjoint regions (Lueh et al. [55]). The idea of the fusion allocator is to delay spilling until the interference graphs of two simplifiable regions get fused. When the combined graph would be no longer simplifiable, the fusion operator, based on feedback profiling information, attempts to split live ranges in order to minimize spill code at the region boundaries.

Gupta et al. [33] proposed clique separators for partitioning the interference graph. A clique separator is a clique that partitions a graph into two disjoint components. The clique allocator computes spans (definition-use chains) and identifies a set of clique separators. Each span can be contained in at most a fixed number of sub-graphs. Each sub-graph is colored separately, while it includes the nodes of a separating clique. The final graph is composed from the sub-graphs possibly with renumbering of assigned registers and spilling (or register copies) at separator boundaries. Given n nodes and m clique separator the clique separator consumes $O(n^2 / m^2)$ space and $O(n^2 / m)$ time.

Splitting the candidates is often implicit in the coloring heuristics. Well-known examples include coloring basic block local candidates first, then the global candidates (e.g. Briggs [12]).

3.3.3 Coalescing

Chaitin's coalescing is aggressive and can yield un-colorable interference graphs. This problem can be avoided by conservative coalescing (Briggs [12], [13]): the live ranges S and T representing the source and destination of a move instruction respectively are combined when they don't interfere **and** when the fused node ST has fewer than K neighbors of significant degree. This rule ensures that the fused node does not trigger a new blocking state in the simplification phase. Therefore conservative coalescing cannot transform a colorable into an un-colorable graph. Iterative coalescing is an attempt to make the conservative method more effective. An allocator with iterative coalescing interleaves "simplification" and "coalesce". Simplification removes only nodes that are non-move related. Here a node is non-move related if it is neither the source nor the destination of a move instruction. In a blocking state "simplify" has only high degree or move-related nodes. It calls "coalesce", which applies conservative coalescing. When at least two nodes can be coalesced, simplification continues. Otherwise a low-degree move-related node is marked as non-move related ("freeze"), so simplification can find a low-degree node and transitions out of blocking state. If no node could be freezed, "spilling" must unblock "simplify". Optimistic coalescing (a delay technique like optimistic coloring) allows conservative coalescing, but undoes the coalescing decision if the simplification phase can’t leave a blocking state because the graph has become uncolorable.

Biasing, introduced by Briggs ([12], [13]), is an opportunistic and non-intrusive method for fusing two nodes. For a single live range L, the sources or destinations in move instructions that reference L are collected in a partner list for L. When “select” colors L, it checks if a color can be used that has been assigned to a partner. If successful, biasing can coalesce nodes when conservative coalescing could not be applied, while creating little time and implementation overhead. In other words, biasing expresses a coalesce preference. The actual coalesce decision is delayed and made at coloring time.

3.3.4 Extensions

Nickerson [60] introduces the concepts of weighted degree and asymmetric interference matrix to handle “cluster” register candidates. Cluster register candidates must be assigned 2 or more register, dependent of the size of the cluster. Additional constraints are that the assigned register “cluster” must be aligned (e.g. first element in cluster must start at a register number divisible by cluster size) and consist of adjacent registers (e.g. when register rx is assigned to the first cluster element, rx+1 must be assigned to the “mate” in a pair cluster). The key observation is that modeling interferences with edges for each element in the cluster would yield conservative results. Instead the interferences must be “normalized” with respect to the first element in the cluster. Smith et al. developed a more general colorability criterion that can handle clusters and overlapping (“alias”) registers. But their allocator could give more conservative results than Nickerson’s method.

3.4 Alternative Models

The probabilistic allocator (Proebsting and Fischer [63]) generalizes “furthest next use” of a variable to a probability. The probability prob(v) of variable to stay in a register at a given program point is approximately the inverse of the distance to the next use. When the probability is low, the live range may get split at this point. The allocator has three phases, local and global allocation, and assignment, which uses a coloring allocator. The global allocation phase proceeds from inner loops to outer loops, where allocation is based on probabilities. Every time a global live range is allocated a register, the probabilities get recomputed, because “probabilities and allocations interact”. Global allocation iterates until the probability that more live ranges can be assigned registers is zero. For a prototype implementation the authors report a 100x slowdown on a sample of six small test kernels. This approach demonstrates the cost of register allocation by attempting to steer allocation one node at a time.

Linear scan methods[1] project live ranges onto a line. The blocks of the CFG are put in a linear order, instructions of the blocks are numbered consecutively and live ranges are sorted in ascending order of their first instruction. A scan line moves sequentially through the ordered set of live ranges and assigns them registers or spill slots as soon as they are hit by the scan line. A best-fit first-end allocator can find an optimal coloring for the ranges in linear time in a single scan. Simplicity comes at the loss of structure, since the projections may generate overlaps that don’t exist in the original control flow graph. This can happen (only) when a live range L1 is contained entirely in a life time hole of a live range L2. A linear scan allocator can model life time holes, but at the cost of its simplicity. In general, linear scan allocators are less powerful and produce slower code than coloring allocators: a linear scan allocator assigns (“locally”) a register immediately when the scan line hits a live range without considering all interferences (“globally”) like the coloring allocator. But they can provide good allocation results at lower compile time compared to a coloring allocator.

Other methods model register allocation as a mathematical programming problem applying (integer) LP (examples are Goodwin and Wilken [32], Fu and Wilken [29], Appel and George [4]) and quadratic solvers (Scholz and Eckstein [68], Hirnschrott et al. [36]). The benefit of these methods is modeling accuracy and optimality of the solvers. Therefore these methods target mainly, but are not limited to, “irregular” architectures. Irregularity means constraints. Examples for constraints include partial register usage and register alignments for groups of candidates. For example, on Itanium a pair of floating-point register must be allocated to consecutive registers fi, fi+1. The benefit of modeling accuracy seems to be offset by low scalability and long compile times. But common irregularities can be modeled in graph-coloring allocators as well (e.g. Nickerson [60] or Smith et al. [72]). To the best of our knowledge there is no study that compares the

[1] This explanation of linear scan methods is from Prof. Mössenböck.

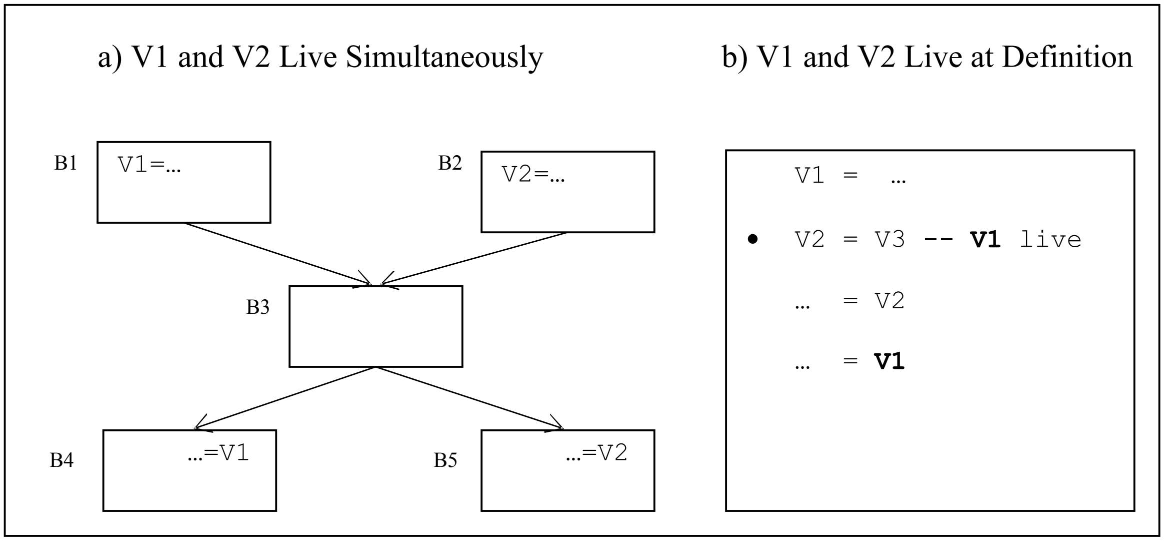
methods (linear scan, graph-coloring, and mathematical models) in a detailed cost/benefit analysis across a rich set of applications and architectures.

3.5 Theory

This section reviews the basic foundation of coloring algorithms. In general, the register allocation problem is equivalent to finding a K-coloring of a graph and is NP-complete when K > 2. In this case there is a polynomial algorithm that verifies a solution, but no polynomial time algorithm exists (unless P = NP) that can decide if an arbitrary interference graph is K-colorable for a fixed K. When K is 2, a linear algorithm that decides 2-colorability exists. In this case the interference graph must be a bipartite graph. For special interference graphs, linear or polynomial algorithms are known. There is a hierarchy of structure: for interval graphs optimal linear time algorithms exist, and for perfect graphs polynomial time algorithms are known. Hack et al. [34] show that interference graphs derived from SSA-form are chordal. Chordal graphs are a sub-family of perfect graphs that contain interval graphs. Like for interval graphs optimal efficient linear algorithms for coloring chordal graphs exist. But in general the interference graph does not exhibit enough structure for a polynomial time solution (unless P = NP). In fact, the exploit of structure is the red thread in the register allocation literature. This can be seen on many examples like containment graphs, preference graphs, live range splitting etc., which can improve allocation results. It seems to be the quintessence of an NP-complete problem that the information to determine the next step towards its solution in any solver is equivalent to finding the solution itself. Attempts to add structure to the interference graph to make better allocation choices (“solution step”) can consume lots of energy (=compile time) for no clear gain. Perhaps the probabilistic allocator (Proebsting and Fischer [63]) is an illuminating example.

3.5.1 Definitions of Interference

The key property of interference is that two interfering live ranges cannot be assigned the same register. Register allocation literature uses two different definitions of interference, “definition” and “simultaneous” interference. Both are conservative, because they can result in allocating more registers than necessary. For Chaitin two live ranges interfere when one is live at the definition of the other (or vice versa). The alternative is that two live ranges interfere when they are live simultaneously at any program point. Both definitions can be shown to be equivalent for strict programs. In strict programs a variable is defined on every path from function entry to a use. For non-strict programs, two live ranges that are live simultaneously do not need to interfere in Chaitin's sense. In Figure 20 there are 5 basic blocks forming a cross and two live ranges, V1 and V2, which are clearly live simultaneously. But, when the control flow graph is acyclic, V1 and V2 do not interfere based on Chaitin's definition, since V1 is not live at the definition of V2 and vice versa. But there are also cases when definition interference is too conservative: when two live ranges interfere in Chaitin's sense, they can still be assigned the same register when they hold the same value at all points of interference. "Value" interference could be relevant to reduce register pressure for programs in SSA-form, where every variable has a single definition.



a) V1 and V2 Live Simultaneously

b) V1 and V2 Live at Definition

```
  V1 =  …
• V2 = V3 -- V1 live
  …  = V2
  …  = V1
```

Figure 20 Definitions of Interference

3.5.2 Coloring Inequality

Let $IG = (V, E)$ be an interference graph with a set of nodes V (variables, live ranges) and a set of interference edges, E.

Let Maxlive be the maximal number of simultaneous live variables at any program point.

Let $\chi(IG)$ be the *chromatic number* of IG, and $\omega(IG)$ be the *clique number* of IG.

Let $M = Max(Degree(v)), v \in V$.

Then the following coloring inequality holds:

$$\boxed{Maxlive \leq \omega(IG) \leq \chi(IG) \leq M + 1}$$

Equation 4 Fundamental Coloring Inequality for Strict Programs

The coloring inequality is folklore. It defines the lower and upper bounds for the colors needed for coloring the interference graph "IG". Perfect graphs are defined by $\omega(IG) = \chi(IG)$. Chordal graphs have the additional property $Maxlive = \omega(IG)$ (Bouchez et al. [10]). In non-strict programs the leftmost inequality is not necessarily valid.

4 The Intel Itanium Compiler

The Intel Itanium C/C++/Fortran compiler employs state-of-the-art analysis and optimizations. It incorporates inter-procedural data analysis and optimizations (IPO), high-level optimizations (HLO) focusing on loop and data transformations, a global scalar optimizer (IL0) and an optimizing code generator (ECG). All optimization techniques can be profile-guided and tuned for the Itanium architecture. This chapter gives a high-level overview of the code generator, develops an intuitive understanding of modulo scheduling and predicates, and reviews the register allocators.

The first phase of ECG, translation, converts the optimizer's intermediate program representation (IL0) into a low level intermediated representation (EIL), which models closely Itanium instructions. The four major optimizations in the code generator are the software pipeliner, the predicator, the global scheduler and the register allocator (Figure 21).

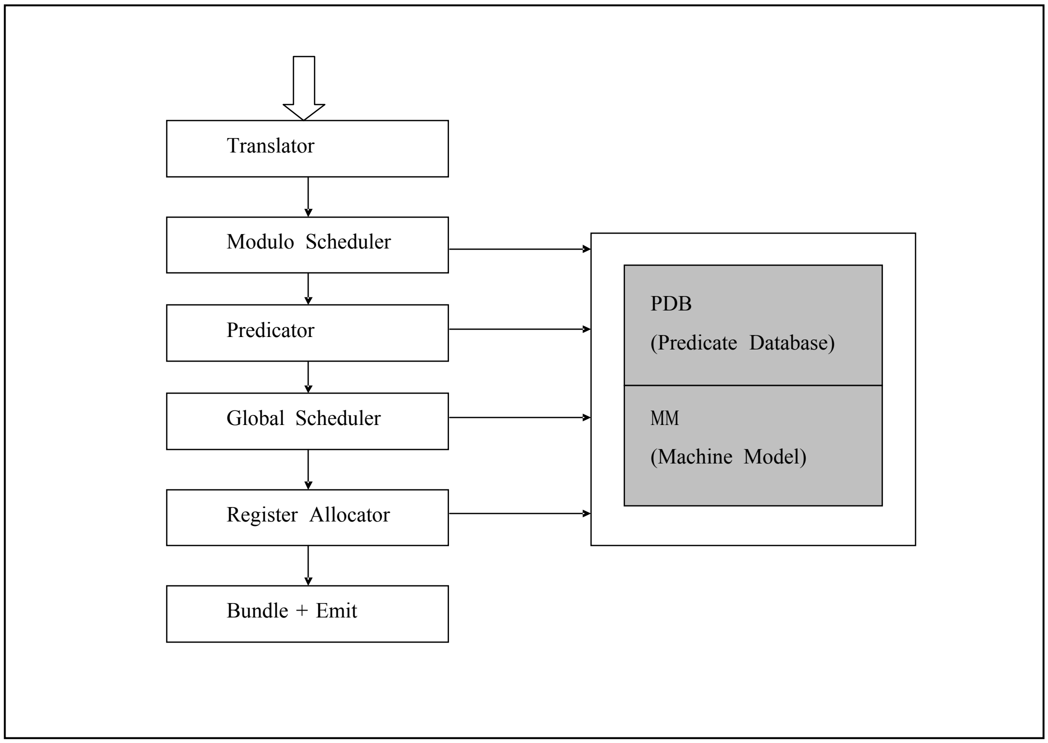


Figure 21 Components and Flow of Itanium Compiler Backend

The four major phases interface with the machine model, which describes in detail the microarchitecture, and a predicate database, which contains relations between qualifying predicates. The most important relation is predicate disjointness. Two predicates are disjoint when they cannot both be true at the any point in the program. In this respect pipelined loops are special. Associated with each software pipelined loop is a predicate disjointness vector. It has the predicate disjointness information for all qualifying (block) predicates in the loop, which is used during modulo scheduling. For non-pipelined code a single interface, the predicate query system (PQS) is used.

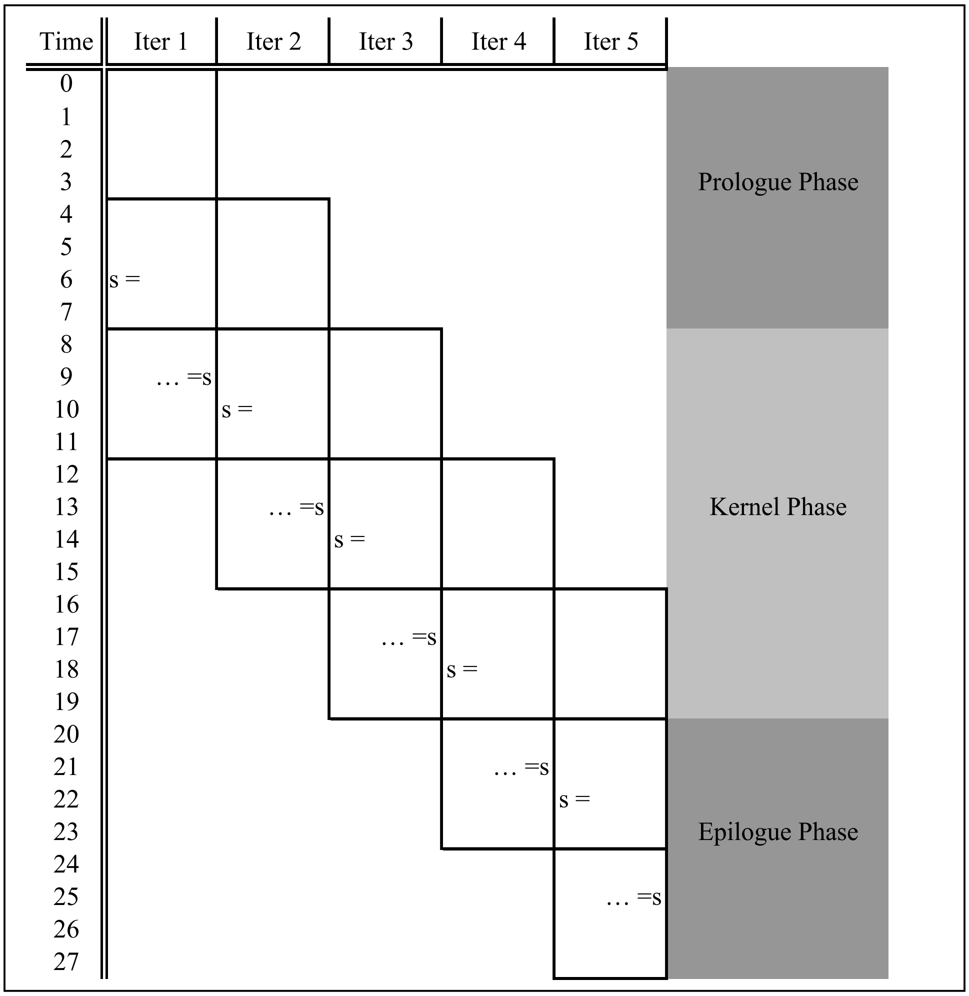


Figure 22 Five Iterations of Pipelined Loop with three Stages and II=4

Modulo scheduling is a loop scheduling technique modeled after a hardware pipeline with SC ("stage count") stages. Each stage has a height of II ("Initiation Interval") cycles.

It takes SC-1 stages to fill the pipeline. Filling happens in the Prologue phase , which starts a new iteration starts every II cycles. Symmetrically it takes SC-1 phases to drain the pipeline. Draining happens in the Epilogue phase, which ends a single iteration every II cycles. In steady state (Kernel phase) the pipeline is full and SC iterations execute in parallel. Every II cycle a new iteration $N > SC$ starts, and iteration $N - SC$ ends. Figure 22 illustrates the concept of software pipelining with its three phases: Prologue, Kernel and Epilogue. It shows a loop with single iteration schedule length of 12 and five iterations (Iter 1, …, Iter 5) of the loop. We assume the loop can be scheduled with an II of 4. Therefore the loop has a stage count SC=3 (=schedule length/ II = 12/4). In the Prologue phase every II (=4) cycles a new iteration starts. In steady state (Kernel phase) the pipeline is full: there execute 3 stages in parallel. Each stage belongs to different loop iteration. Pipelining is a throughput optimization. In the Kernel phase SC iterations execute in parallel. Assuming we execute 5 iterations, the Epilogue phase starts at cycle 20. In the Epilogue phase one loop iteration ends every II cycle.

We briefly explain the relation between one loop iteration and the module schedule of a loop. Figure 23 shows one loop iteration and modulo schedule of a three stage loop with an II of 4. Each stage has its own stage predicate starting with p16, which is the first rotating predicate register. p16 is the stage predicate for the first stage, p17 for the second stage and p18 for the third stage. The stage predicates control the phases of the pipelined loop. Before execution of the pipelined loop p16 is set. So when the modulo scheduled loop runs, all instructions of the first stage of the first loop iteration execute. The loop branch instruction then sets p17 in addition to p16. In the second run the instructions in the second stage of the first loop iteration and the first stage of the second loop iteration execute. The next loop branch sets p18 also and execution of the pipelined loop leaves the Prologue phase and enters the Kernel phase[2]. Entering the epilogue phase, p16 is cleared first, (=set to False) then p17 etc. The single iteration schedule length is II times SC. The II is determined by two factors: machine resources available and data recurrences. In the pipelined loop at time T (referred to as "Modulo" in Figure 23) instructions of all stages with cycle time % II = T may execute in parallel (in the Kernel phase) and may not exceed machine resources available per cycle ("resII"). For example,

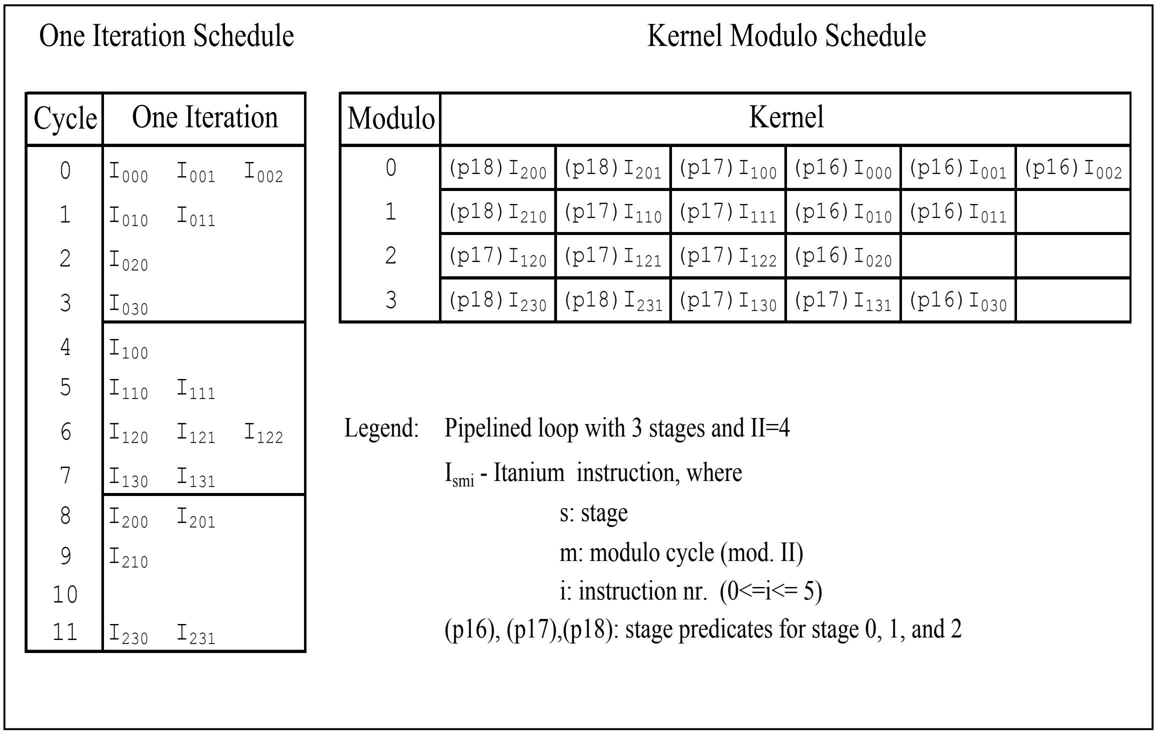
[2] It is possible that a pipelined does not enter the Kernel phase and still give performance benefits

when T=1, instructions of cycle 1, 5 and 9 from the single iteration schedule execute in parallel in module schedule (in the Kernel phase). In addition, the II must be large enough to cover the maximal dependence distance of all loop data, so the following equation must hold:

II * (iteration) distance >= maximal dependence cycle length (“recII”)

When a value is computed in iteration k and used in iteration k+n, the recII guarantees that there are n loop iterations to cover the dependence distance between the definition and the use. Combining both, resource and recurrence, constraints results gives Min II = Max (resII, recII), where MinII is the best II possible.

IA-64 supports rotating registers, predication, and special loop branches (br.ctop, br.cexit, br.wtop, br.wexit) to enable kernel-only innermost modulo scheduled loops, where the kernel contains the prologue, steady state and epilogue phases controlled by stage predicate and state registers like ar.lc (loop count) and ar.ec (epilog count).



One Iteration Schedule

Cycle	One Iteration		
0	I_{000}	I_{001}	I_{002}
1	I_{010}	I_{011}	
2	I_{020}		
3	I_{030}		
4	I_{100}		
5	I_{110}	I_{111}	
6	I_{120}	I_{121}	I_{122}
7	I_{130}	I_{131}	
8	I_{200}	I_{201}	
9	I_{210}		
10			
11	I_{230}	I_{231}	

Kernel Modulo Schedule

Modulo	Kernel					
0	(p18) I_{200}	(p18) I_{201}	(p17) I_{100}	(p16) I_{000}	(p16) I_{001}	(p16) I_{002}
1	(p18) I_{210}	(p17) I_{110}	(p17) I_{111}	(p16) I_{010}	(p16) I_{011}	
2	(p17) I_{120}	(p17) I_{121}	(p17) I_{122}	(p16) I_{020}		
3	(p18) I_{230}	(p18) I_{231}	(p17) I_{130}	(p17) I_{131}	(p16) I_{030}	

Legend: Pipelined loop with 3 stages and II=4

I_{smi} - Itanium instruction, where

s: stage

m: modulo cycle (mod. II)

i: instruction nr. (0<=i<= 5)

(p16), (p17),(p18): stage predicates for stage 0, 1, and 2

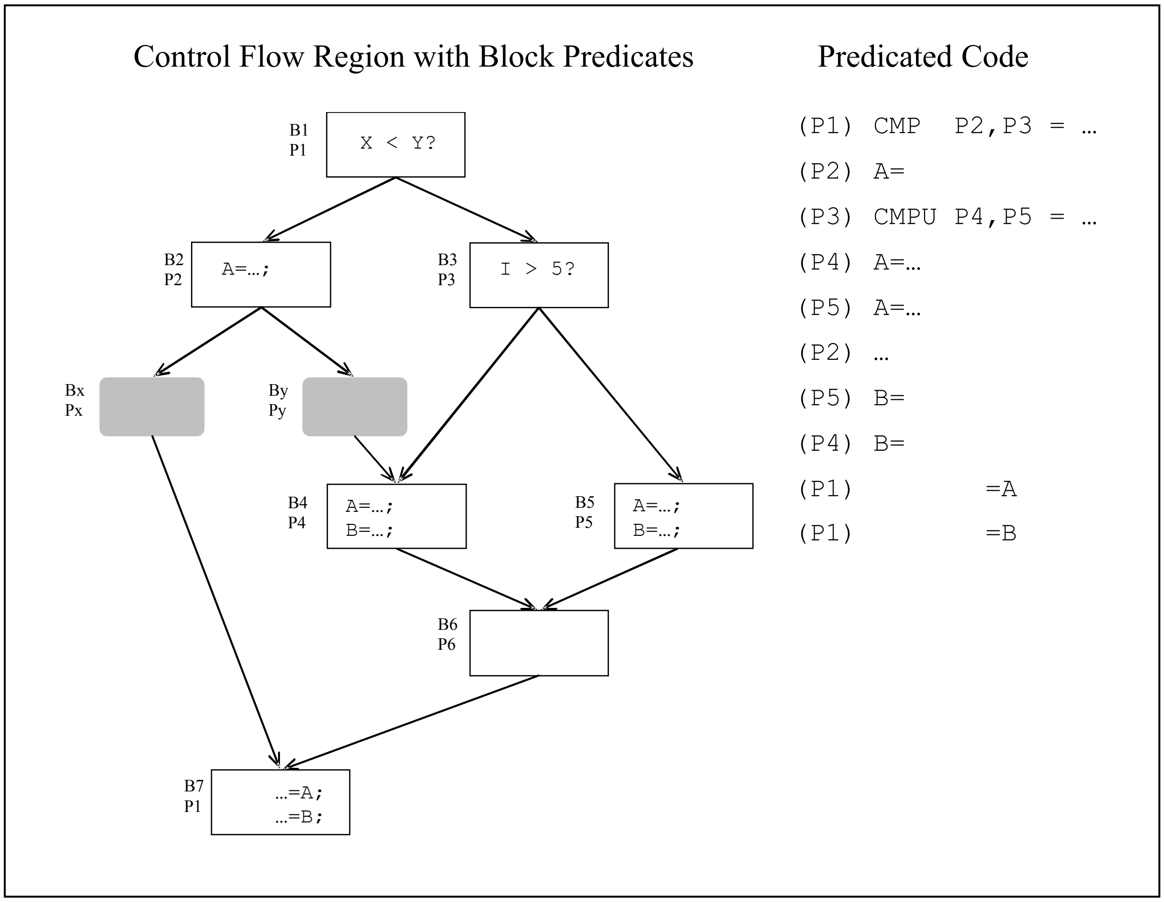
Figure 23 One Iteration and Kernel Schedule for a Loop

As for register allocation, live ranges that span multiple stages of a pipelined loop are allocated to rotating registers. The allocation of these live ranges to rotating registers is handled by the software pipeliner. There is one uncommon feature. Since the rotating registers are multiples of 8 on the register stack, the pipeliner uses them carefully. It leaves small live ranges that have a length less than II to the register allocator. As long as the live range is contained within a pipeline stage, all references are under the same stage predicate. But it can happen that a small live range crosses a stage. Then the definition and use are under different stage predicates. For example, in Figure 22 symbolic register s is defined in cycle 6 of stage 2 and used in cycle 9 of stage 3. The dependence distance is zero, since the definition and use are within the same iteration. Since the length of the live range of s is 3 (9-6=3), its length is less than II and it is left for allocation to the coloring allocator. Unless the coloring allocator is pipeline aware, however, the references to s in the 'modulo' schedule will look as if there is a use of s before there is a definition. The reason is this: the use is in cycle 9. The 'modulo' cycle for the use is 1 (9%4=1). On the other hand, the 'modulo' cycle for the definition is 2 (6 %4=2), so the use appears before the definition in the 'modulo' schedule. Applying live variable analysis naively would allow the live range of s to escape to the outermost loop that contains the pipelined loop. Since cross stage live ranges are well defined, the live variable analysis recognizes them and ensures that they cannot escape the pipelined loop. It will stop live propagation into the predecessor block of the pipelined loop, but ensures the live range is live across the loop back edge.

Predication is the conditional execution of an instruction guarded by a qualifying predicate. Using predication the compiler can merge multiple control flow paths into a single predicated region. Predication can remove mispredicted branches, may decrease code size, increase code motion flexibility, remove control flow in pipelined loops and can increase ILP. Cost of if-conversion includes a possible increase in code size, an increase in schedule length and the potential for a mere transfer of a misprediction rather than its removal.

The predicator in the compiler performs if-conversion in a series of steps. In a preparation step it assigns a predicate to each block using the RK algorithm [62], where R and K are the names of mappings used to determine control flow equivalence. The characteristic of the RK algorithm is that control-dependent equivalent nodes get assigned the same predicate. The next steps are region picking, benefit evaluation and if-conversion: First, it picks a region. In the Intel compiler all candidate regions are single entry and multiple exit regions. Second, it evaluates the benefits of if-converting the region by estimating the execution times for the predicated and unpredicated (=control flow region). The estimates take the machine resources, schedules and feedback profiling data (static or dynamic) into account. Third, it materializes the predicates by generating compare instructions. Each instruction in the predicated region is predicated with the basic block predicate. The materialized block predicate becomes the qualifying predicate of the instruction. An example is in Figure 24. It shows a snippet of the control flow graph, which is a single entry – single exits region. Basic blocks B1 and B7 are control-dependent equivalent and get assigned the same block predicate, P1. The edges from block B2 to B4 and from B2 to B7 are critical, because B2 has more than one successor and each successor (B4 and B7 respectively) have more than one predecessor. The completion phase inserts new basic blocks Bx and By with predicates Px and Py respectively. The predicated code in this case is a basic block. In the general case, when the control flow graph region is a superblock with multiple exits, the predicated region is a hyperblock. All paths in the original control flow graph overlap and correspond to execution traces in the hyperblock. The instructions in the hyperblock are guarded by qualifying predicates. Proper predicate initialization ensures that each path in the control flow graphs maps 1:1 to an execution trace in the predicated code. An execution trace in the predicated code is a set of predicated instructions, where the qualifying predicate is set. For predicate initialization IA-64 provides unconditional compare instructions (exemplified by CMPU in Figure 24) together with regular compare instructions. When the qualifying predicate of an unconditional compare is clear, the predicate destinations are initialized with zero. Otherwise an unconditional compare behavior matches corresponding regular compares.

The global code scheduler picks acyclic regions for scheduling. Like the predicator it can nest scheduled regions and hoist instructions across regions. The code scheduler generates control- and data speculated code as well as predicated code. It is a major phase in the compiler and responsible for extracting ILP. It is well described in [9] [8].



Predicated Code

```
(P1) CMP  P2,P3 = …
(P2) A=
(P3) CMPU P4,P5 = …
(P4) A=…
(P5) A=…
(P2) …
(P5) B=
(P4) B=
(P1)        =A
(P1)        =B
```

Figure 24 Control Flow-and Predicated Region

The high-level architecture of the register allocator is in Figure 25. The input to the register allocator is local and global virtual registers, as well as physical registers. A virtual register is local if all its references are confined within a basic block. All virtual registers have a unique number. All live ranges are coherent. They cannot be split into disjoint components. Non-coherent live ranges are split into their disjoint parts and renamed before allocation.[3] The allocator is region based. There can be multiple region entries and exits. Each region is allocated invoking a Chaitin-style coloring allocator. Allocations are reconciled at region boundaries. At region transitions reconciliation must follow store (register – memory), move (register – register) and load (memory – register) order. For example, if a candidate resides in a register in region 1, in memory in region 2

[3] With the exception of speculated live ranges

and transition is from region 1 to region 2, then the store must happen before any reconciliation move or load of any other candidate. There are two allocation steps: first, floating-point, predicate, and branch registers are allocated. Spilling such register candidates introduces integer register candidates. Second, the integer registers are allocated. After the allocations, memory stack layout is finalized. Register stack layout is determined during allocation, but the alloc instruction is issued in a later phase. The following describes the features of the register allocator in terms of the categories of Section 3.3, although the actual implementation may differ from the descriptions.

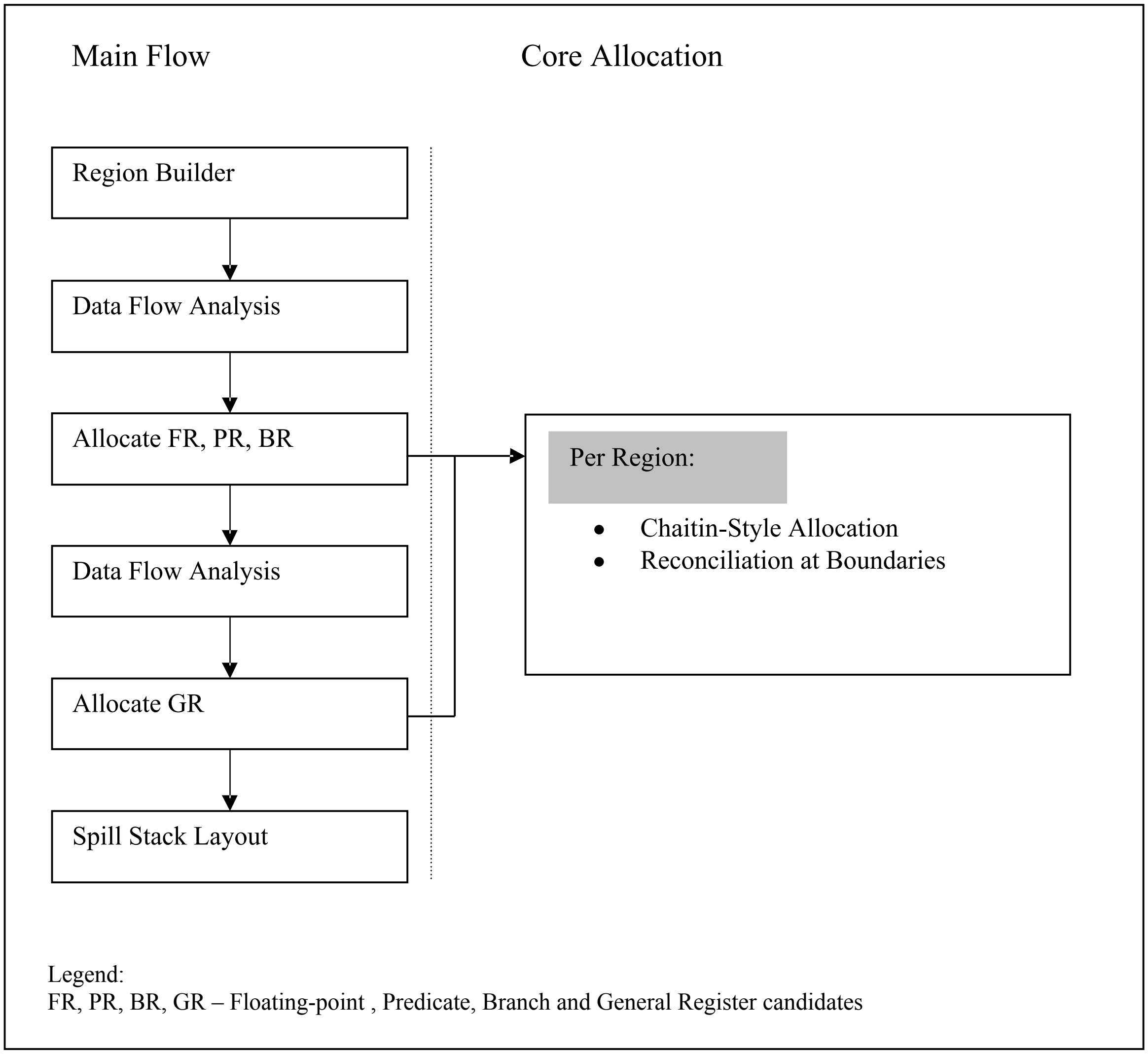


Figure 25 Register Allocator Architecture

Spill code reduction:

- The allocator supports Chaitin's heuristics, rematerialization, and pre-materialization. It has benefit –driven simplification for speculated live ranges and a set of heuristics for ordering candidates on the coloring stack, although not all candidates are pushed onto the coloring stack. The assignment phase is also benefit-driven and computes benefits for assigning scratch, preserved or stacked registers.

Scoping:

- The global register allocator is region-based and allocation starts with the innermost loops. Pipelined loops are allocated first. When there is no loop in the control flow graph, the entire function is the only allocation region. All candidates in the region are mapped to physical registers. The mapping is stored as a global register preference, which subsequent allocations try to honor.
- The inter-procedural register allocator is opportunistic. Register usage is recorded. The caller avoids spills of scratch registers at call sites that don't use this register, if the live range contains the call.

Extensions:

- The register allocator is predicate-aware, pipeline-aware (pipelined loops), speculation-aware and employs efficient allocation schemes for the register stack. Itanium has load pair instructions requiring even-odd or odd-even allocation for the load pair destinations, which the allocator takes into account.

Coalescing:

- The allocator implements a form of biasing. It has associated with each candidate a set of prioritized preferences.

In addition the code generator has a local register allocator that can allocate register candidates within a basic block. It can get invoked for functions that have huge memory requirements because of many candidates. Such functions are usually generated by program generators in large server applications. Since memory requirement of the interference matrix grows quadratic with the number of register candidates, a routine with e.g. 200000 register candidates would require more than 0.3 Gb of memory just to represent the interference matrix. In huge functions, memory consumption is the reason for long compile times of a coloring allocator.

5 Exploring the Register Stack

On IA-64, each procedure has its own variable size register stack (Figure 5) with its own variable number of rotating registers. The *alloc* instruction specifies the register stack frame of a procedure: the number of incoming parameters (in), the number of local registers (loc) and the number of outgoing parameters (out). The total number of registers in a register stack is in+loc+out <= 96. Usually a register stack frame has up to 8 incoming argument register and 8 outgoing parameter registers. The architectural registers ar.pfs contains fields that describe the frame of the caller ("previous function state") and is saved into the destination register of the alloc instruction for register stack unwinding. In the standard code generation model the compiler issues a single alloc instruction to create the register stack frame. The drawback of this method is that the frame must be large enough for the entire procedure. For example, if a procedure has two different calls with 2 and 8 outgoing parameters respectively, the alloc instruction needs to reserve 8= max(2,8) outgoing registers. This example can be generalized: when the alloc instruction reserves more stacked registers than the dynamic execution trace uses, unnecessary spills to the backing store (Section 2.5, p. 26) could result. IA-64 allows resizing the register stack by using more than one alloc instruction within a function ("multiple alloc", Hoflehner and Pierce [38], Settle et al.[69], Hoflehner et al.[39]). Again, the key here is that alloc instructions in different functions on the call stack increase the register stack, but multiple alloc instructions within a single function only resize its register stack frame, but do not add or remove frames. The effectiveness of the method is demonstrated in the example in Figure 26 (where the ar.pfs and alloc destination register are omitted). The register stack is at first increased by two functions on the call stack, foo and foo1, causing spilling to the backing store. Originally, the compiler generates a single alloc instruction (alloc rx=1,2,1,0) in foo and foo1. In the assembly listing we increase the number of stacked registers so that the back to back call of foo and foo1 incurs RSE spills. In the last step we shrink the register stack before the calls to foo1 and foo2. Inserting multiple alloc instructions that shrink the stack before any RSE traffic can be triggered gives a 6.5X speed-up on Itanium compared to the version with a single alloc instruction. The performance of the multiple alloc versions matches the original, unmodified version.

Code Snippets

```
foo2(int i) { return; }
foo1(int i) { foo2(i+2); }
foo(int i) { foo1(i+1); }
main() { int i; for (i=1; i < 10000000; i++) foo(i);}
```

Original and modified register stack for foo() *and* foo1():

```
Original          with Increased Frame    with Multiple Alloc

alloc rx=1,2,1,0  alloc rx=1,50,1,0       alloc rx=1,50,1,0
                                          …
                                          alloc ry=1,2,1,0

                                          br.call {foo1|foo2}

                                          alloc ry=1,50,1,0
```

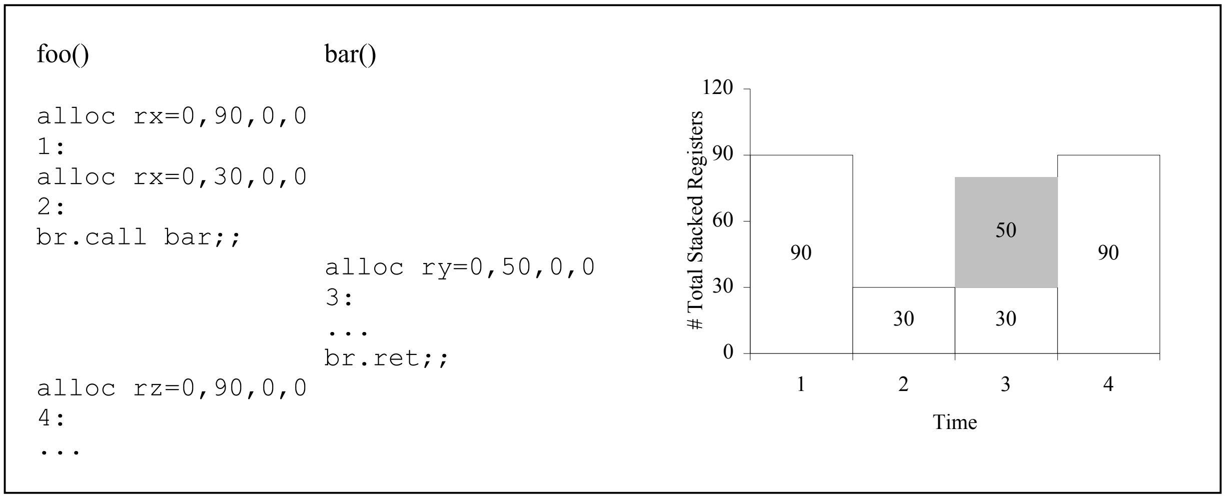
Figure 26 Proof of Concept to show Effectiveness of Multiple Alloc

More details about register stack resizing are in

Figure 27. The register stack is shrunk before a call site and restored to its original size afterwards. This can reduce the total number of registers consumed by the caller and callee and consequently the overall RSE traffic for the application. In the example procedure foo uses 90 stacked registers. At the point of the call to bar, 60 registers on the stack are found dead and the register stack gets resized accordingly. Procedure bar uses 50 stacked registers. The combined register stack of foo and bar uses 80 registers. It fits into the register file and spilling to the backing store is avoided. Without resizing the combined register stack would be 140=90+50 registers, resulting in 46 = 140 – 96 spills to the backing store. Dead registers on top of the stack are determined by live range analysis. If the number of dead registers on top of the register stack exceeds a given threshold, the register stack is reduced by the amount of dead registers before the call. Parameter registers have to be remapped so that they stay on top of the resized register stack. This parameter mapping is not shown in the example. There is no principal

difficulty. Since the register allocator does not know the size of the register stack, it cannot pre-determine the parameter (=out) registers. Therefore it assigns symbolic registers which are mapped to the proper physical registers in a post-pass. With multiple alloc this routine has to determine the actual size of the register stack at the call site.

The optimization is opportunistic in the sense that the compiler cannot have a perfect knowledge of the state of the RSE when it inserts the extra alloc instructions. Specifically, the compiler does not know if the reduction of the register stack will actually decrease the RSE traffic at run-time. On the other hand, the cost of the optimizations is extra alloc instructions, which have scheduling constraints (e.g. alloc must be the first instruction in an instruction group) and may contribute to an increase in code size.



```
foo()                 bar()

alloc rx=0,90,0,0
1:
alloc rx=0,30,0,0
2:
br.call bar;;
                      alloc ry=0,50,0,0
                      3:
                      ...
                      br.ret;;
alloc rz=0,90,0,0
4:
...
```

Figure 27 Shrinking Register Stack before Call

Multiple alloc can be effective only on paths that contain a call or a return. The algorithm in the Intel compiler forms regions of different register pressure after register allocation. Extra alloc instructions are inserted at region boundaries when the difference of the register stack sizes exceeds a threshold. Candidates for regions with high register pressure are software pipelined loops, since many loop iterations can execute in parallel. In the Itanium compiler rotating register allocation is handled by the pipeliner. Register lifetimes that fit within II are passed on to the graph-coloring allocator. The pipeliner controls the numbers of interferences and checks the availability of registers with a register server. This guarantees spill free code in pipelined loops, since a) pipelined loops get allocated first and b) the pipeliner ensures that sufficient registers are available to allocate all candidates.

The rotating registers within a procedure cannot simply be assigned to global virtual registers that span software-pipelined loops. Consider live range V1 in Example 3, which is defined before (line 2) and used after the loop (line 7). Assume the first loop executes eight times and r40 (r32-r39 are used in the first loop, which uses eight rotating registers) is assigned to V1. Then the register allocator would have to perform context-sensitive register renaming to ensure correctness. After the swp loop r40 has to be accessed as r48 because of register rotation. For this reason the allocator avoids such assignments to live ranges spanning pipelined loops. It also has to assume that such a live range interferes with all rotating registers. This is different from usual interferences since rotating registers can be implicit. For example, in the loop only r32 may be visible. But, with 8 registers rotating, r32 represents all registers r32-r39, so registers r33-r39 are implicit and not directly 'visible' for the allocator.

```
                .proc foo
  1:            V2 = alloc ar.pfs, 2, 88, 2, 32
  2:            mov V1=           // mov r40
  3:  prolog1:  ...
  4:  loop1:    // swp loop with 8 rotating registers
  5:            br.cloop loop1
  6:  epilog1:  ...
  7:            add   V3=V4,V1    // add r41=r44, r48
  8:            ...
  9:  prolog2:  ...
 10:  loop2:    // swp loop with 32 rotating registers
 11:            br.ctop loop2
 12:  epilog2:  ...
 13:            br.ret
```

Example 3 Two Pipelined Loops in Single Alloc Routine

There is another unusual aspect of pipelined loops. They need multiple alloc to remain spill free after register allocation. In single *alloc* procedures, the code generator has to specify the maximal number of rotating registers any swp loop in the procedure demands. In Example 3 the code generator specifies 32 =Max (8, 32) rotating registers in the *alloc* instruction, since the first loop uses 8, while the second loop requires 32 rotating registers. Therefore the pipeliner may underestimate the register pressure for the candidates in the first loop that in fact uses only 8 rotating registers. Since the pipeliner works on one loop at a time, it is unaware of register requirements of subsequent loops. For example, with 8 registers, the pipeliner assumes 88=96-8 registers available for the allocator. But since the alloc instruction must specify 32 rotating registers, de facto the allocator has only 64=96-32 registers. This can result in spills in pipelined loops with a significant (run-time) performance cost.

```
procedure insert_alloc_instr(BLOCK P, INT r)
     before first definition of
     physical rotating register in B:
          insert alloc rx=ar.pfs, i,l,o,r;
endproc
procedure insert_clrrrb_instr(BLOCK B)
     after last use of a physical rotating register
     in B:
          insert clrrrb
endproc
procedure ma_for_swp_loops
     foreach software_pipelined_prolog P
          r = #rotating registers in pipelined loop;
          insert_alloc_instr(P, r);
     endfor
     foreach software_pipelined_epilog E
          insert_clrrrb_instr(E);
     endfor
endproc
```

Algorithm 1 Multiple Alloc for SWP Loops

Spilling in pipelined loops can be avoided with multiple allocs. An alloc instruction can resize the number of rotating registers [13 Vol. 1, p. 4.2]. This can be used by the register allocator to adapt to the needs of rotating registers in a pipelined loop. In procedures that contain multiple swp loops with various numbers of rotating registers, Algorithm 3 inserts alloc and clrrrb instructions in the prolog and epilog of swp loops. The clrrrb instruction resets the rotating register base. This is usually necessary when dynamically an alloc instruction can follow in the same procedure. Otherwise the RSE could fault at the alloc. This also means that in general a live range spanning a clrrrb instruction cannot reside in a rotating register. Wrapping the pipelined loop with an alloc instruction ensures that the register allocator can match the estimate of the pipeliner and avoid spills in the loop.

Example 4 shows the effect of Algorithm 1. The swp loops are encapsulated by alloc/clrrrb instructions. As only 8 registers are rotating in the first loop, the register allocator can use 26 more registers reducing the register pressure overall in the procedure. Register r40 can be assigned to V1 and r41 can be assigned to V2.

```
                .proc foo
                V2 = alloc ar.pfs, 2, 62, 2, 0
                mov V1=           // mov r40=
    prolog1:    V<dummy> = alloc ar.pfs, 2, 62, 2, 8
    loop1:      // swp loop with 8 rotating registers
                br.cloop loop1
    epilog1:    clrrrb
                add   V3=V4,V1    // add r43=r44, r40
                ...
    prolog2:    V<dummy> = alloc ar.pfs, 2, 62, 2, 32
    loop2:      // swp loop with 32 rotating registers
                br.wtop loop2
    epilog2:    clrrrb
                …
                br.ret
```

Example 4 SWP Loops with Multiple Alloc

There is cost to the extra alloc instructions: the alloc instruction must be the first instruction in an instruction group. It is WAW dependent on a call and needs a result register.

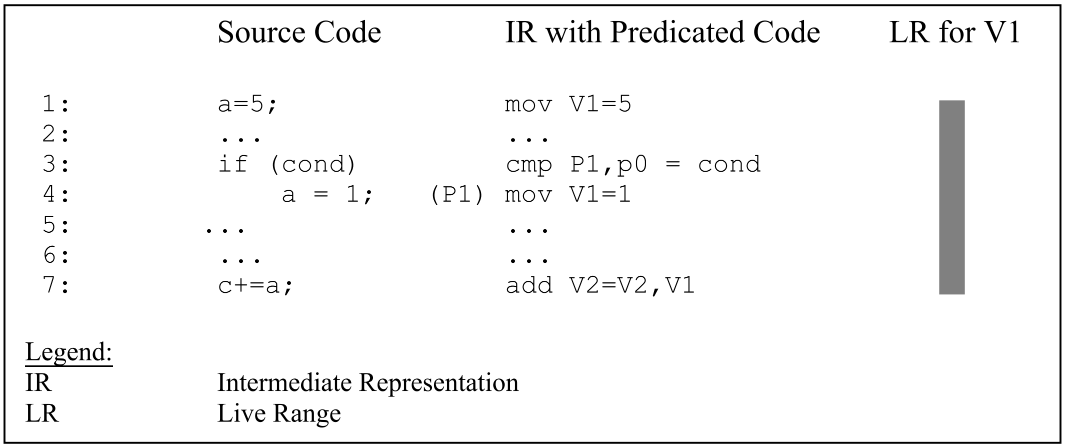
6 Register Allocation for Predicated Code

The IA-64 architecture is a fully predicated architecture with 64 predicate registers [42]. Each instruction (with some exceptions like the *alloc* instruction) is guarded by a qualifying predicate. If the qualifying predicate is clear (= set to zero or False) the instruction is discarded in the write back stage of the processor pipeline. If the qualifying predicate is set (= set to one), the instruction is retired. This means the results are committed to architectural state. A fully predicated architecture supports if-conversion, an optimization that eliminates forward branches (Allen et al. [2]). If-conversion transforms a region of the control flow graph to linear ("predicated") code. In this predicated region all execution paths of the original control flow region overlap. Instructions that were control-dependent on a branch that gets eliminated become data dependent on the qualifying predicates introduced during if-conversion. The compiler picks a single entry acyclic control flow region as a candidate for if-conversion. Basic blocks where the paths originating from the single entry meet, are merge points and mark a potential end node for the region former. The candidate region may have multiple exits, so in general it can be considered to be a superblock. The compiler limits the size of the candidate region by setting a threshold for the number of basic blocks that can be in region. The decision whether or not to if-convert a candidate region is driven by a predication oracle. It computes and compares the estimated execution times of the predicated and control flow version of the candidate region. When the estimated execution time for the predicated region is faster than the estimated execution time for the original control flow regions, the candidate region is if-converted into a hyperblock (Mahlke et al. [56]). The register allocator must handle predicated code formed from control flow graph regions. This chapter investigates the impact of predicated code on the register allocator, discusses the predicate query system (PQS), classifies predicated live ranges and interference tracking, and presents a family of predicate-aware register allocators. In particular the "use-plus-partition" allocator is equivalent to a PQS based allocator.

6.1 Impact of Predicated Code

On a predicated architecture completeness and soundness of live variable and disjoint live range information are harder problems than for unpredicated code. For example, for live variables, completeness means at any program point where a variable is actually live, liveness computation reports it as live. Soundness means that at any point where the liveness computation reports a variable as live, it is actually live. The remainder of this section discusses live range extension, interference graph construction for predicated live ranges and global disjointness information.

In unpredicated code a definition is the start of a live range. This is not necessarily true for predicated code. In Example 5 the predicated live range for virtual register V1, which corresponds to variable “a” in the source code, must span lines 1 to 7. This shows that a predicated definition of V1 (line 4) is not necessarily the start of the live range. Otherwise, the live range for V1 would extend incorrectly from line 4 to 7 in the predicated code.



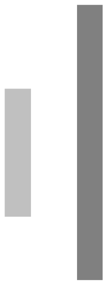
```
            Source Code           IR with Predicated Code       LR for V1

 1:         a=5;                  mov V1=5
 2:         ...                   ...
 3:         if (cond)             cmp P1,p0 = cond
 4:             a = 1;       (P1) mov V1=1
 5:        ...                    ...
 6:         ...                   ...
 7:         c+=a;                 add V2=V2,V1
```

Legend:
IR Intermediate Representation
LR Live Range

Example 5 Source Code, Predicated Code and Predicated Live Range

On the other hand, if no predicated definition is the start of a live range, predicated live ranges would extend to more program points than necessary increasing register pressure and consumption. All predicated live ranges would behave like live ranges of undefined or partially defined variables in unpredicated code. A variable is partially defined if there is (at least) one definition-free path from function entry to a use and (at least) another path that contains a definition. Depending on the run-time execution path the variable is defined or undefined. In cyclic code, the live range of a partially defined variable typically spans the entire loop nest that contains the definition. This is a result of the dataflow algorithms that determine a live range. Available variable analysis (forward) propagates the *is_available* property to every point in the loop nest. Live variable analysis (backwards) propagates the *is_live* property from the use to every point in the loop nest. Thus the live range, which consists of all program points where a variable is both available and live, spans the entire loop nest. This is correct since usually the variable is defined in the first iteration and used in all subsequent iterations. Since the variable is live across the entire loop nest, it interferes with all variables in the loop. Therefore it will not be "destroyed" after being defined in the first iteration.

```
                  Source Code  IR with Predicated Code   LR for V1
 1:               ...               ...
 2:         loop:             loop:
 3:               ...               ...
 4:               if (cond)         cmp P1,P2=cond
 5:               a=2;  (P1)  mov V1=2
 6:               else  a=1;  (p2)  mov V1=1
 7:               ...               ...
 8:               c+=a;       add V2=V2,V1
 9:               ...               ...
10:               goto loop;  br.cond loop
```



Legend:

IR	Intermediate Representation
LR	Live Range

Example 6 Live Range Extension in Predicated Code

In acyclic code, live range extension cannot occur because a live range cannot extend to a program point where it is not available. Example 6 illustrates live range extension in cyclic code for the predicated live range of V1: unless a predicated definition is recognized as the start of the live range for V1, it will extend across the entire loop nest (the large live range from line 2-10). The correct live range is the small live range from line 5, the first predicated definition of V1, to line 8, the use of V1. Live range extension for predicated code could also cause non-termination of a coloring allocator: When the allocator spills a predicated live range, it introduces one or more new predicated live ranges replacing the original. But the new live ranges share the same predicate. Due to live range extension the interferences in the round of allocation may actually increase. Since the new live ranges introduced for spilling are marked as non-spillable the allocator may no longer find spill candidates in the simplification phase. At this point the allocator would have to give up. So there is not only potential performance degradation from extra register pressure, but also a stability reason why a predicate-aware allocator must recognize the start of predicated live ranges. This impacts two building blocks of the allocator: live range analysis (in particular, live variable analysis) and interference graph construction.

In unpredicated code interference is a function of liveness. In predicated code interference is a function of liveness *and* predicate disjointness. Disjoint live ranges do not interfere and can be assigned the same register. The allocator queries a PDB for disjointness information when it constructs the interference graph: if the sets of predicates that guard two live ranges A and B are disjoint, no interference edge needs to be added between them.

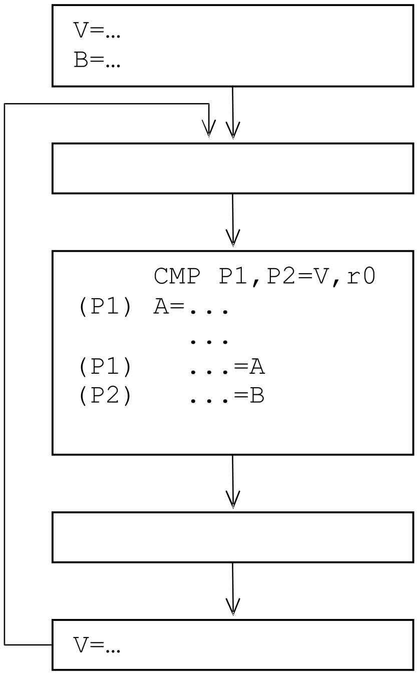
Interference graph construction for predicated code is similar to unpredicated code: it is a backward scan of the instructions in a basic block with a single, unpredicated live vector initialized with the live-at-exit candidates. For each candidate, the predicates under which the candidate is live can be recorded as predicate sets associated with the live range in a separate table. The inference routine takes the qualifying predicate of the current instruction that defines a candidate, the candidate and the live vector as arguments. For each live candidate in the live vector it checks if the qualifying predicate is disjoint from all predicates in its predicate set. Interferences are recorded in the interference graph. Each candidate defined in the current instruction is removed from the active live set when it is the start of the live range. Each candidate used in the current instruction is added to the live vector and the qualifying predicate is recorded in its predicate set. The candidate is live under the predicates in its predicate set. The candidate is dead when its predicate set is empty.

Algorithm 2 shows the code for interference graph construction for predicated live ranges.

```
procedure interfere(PREDICATE qp, VAR v, SET s)
    foreach member s in S
        P = s.predicates;     // SET of predicates
                              // under which
                              // s is live
        if (any p in P and qp are NOT disjoint)
            add_interference(v,s); // not shown.
        fi
    endfor
endproc
procedure add(PREDICATE qp, VAR v, SET S)
    - add v to S and update set of predicates
      under which v is live.
endproc
procedure delete(PREDICATE qp, VAR v, SET s)
    - update set of predicates under which
      v is live.
    if (isEmpty(predicate set of v))
        remove v from s;
    fi
endproc
procedure build_interference_graph()
    foreach basic block B
        L = live_out [B];
        foreach instruction I backwards
            qp = qualifying predicate(I);
            foreach definition d in I
                delete(qp, d, L);
            endfor
            foreach definition d in I
                interfere(qp, d, L);
            endfor
            foreach use u in I
                add(qp, u, L);
            endfor
        endfor // foreach instruction
    endfor //foreach basic block
endproc
```

Algorithm 2 Interference Graph Construction for Predicated Code

Finally, a predicate-aware live variable analysis *must* treat predicated live ranges conservatively across back edges in any practical scenario. For example, in Figure 28 variable B is live under P2 and variable A under predicate P1. In one scenario, P1 could be true in the first iteration and false in the second. If B were live only under P2, the allocator would recognize A and B as disjoint and could assign them the same physical register. In this case the assignment to A in the first iteration would overwrite B, which is used in the second iteration. W hen the live range of B becomes live under p0 (the True predicate) along the back edge, then–since p0 interferes with every predicate–the live ranges for A and B are no longer disjoint.



```
V=…
B=…
```

```
     CMP P1,P2=V,r0
(P1) A=...
       ...
(P1)   ...=A
(P2)   ...=B
```

```
V=…
```

- P1 and P2 are disjoint
- Assume P1 is true in first iteration
- Assume P2 is true in second iteration
- If A and B are assigned same register, then *use* of B under P2 actually uses value of A from the previous iteration.

➔ Propagate liveness property under p0 across back edges. In irreducible graphs propagate liveness property under p0 across retreat edges.

Figure 28 Liveness must be propagated under “p0” across Back Edge

For general graphs, global disjointness can be represented like interference in a triangular matrix of size $O(|B|^2)$, where |B| is the number of basic blocks in the routine. Global disjointness calculation is reaching-definition analysis for block predicates on the acyclic control flow graph. This graph is derived from the original control flow graph by removing back edges. Attention must be paid to irreducible graphs, which have retreat edges that are not back edges. The prototype of an irreducible graph is the triangle graph in Aho [1]. Removing retreat edges gives an acyclic graph, but disjointness is not necessarily consistent with local disjointness, which is based on an arbitrary acyclic region in the graph. Irreducible graphs and disjointness are discussed in more detail in Appendix 12.2.

6.2 Predicate Partition Graph (PPG) and Query System (PQS)

The predicated query system (PQS) is one possible implementation of the PDB referred to in Figure 21. In predicated code the allocator and dataflow algorithms must reason about predicates. The interference computation phase of the allocator and live variable analysis must find the start of a live range. Interference computation must recognize disjoint live ranges. The predicate query system provides predicate information to solve both problems. It is a set of predicate query routines on the predicate partition graph (PPG). The PPG is a directed acyclic graph whose nodes represent predicates and whose labeled edges represent partition relations between predicates. A partition $P = P1 \mid P2$ is represented by labeled edges $P \xrightarrow{r} P1$ and $P \xrightarrow{r} P2$. The common label indicates both edges belong to the same partition (Johnson and Schlansker [45]). The partitions represent execution paths and are derived in one traversal of the control flow graph. The PPG is built by the following rules:

- The start block is assigned the root predicate, e.g. P0. The construction assumes there is always one start block for the CFG of predicate region. The root predicate is unique.
- Successor Rule: For each block in the graph that has two or more successors the partition $P \to P1 \mid P2 \mid ... \mid PN$ is added to the graph, where P is the block predicate of the block and predicates Pi (i=1,…,N) are the block predicates corresponding to the successors ("forward edges $P \to Pi$")
- Predecessor Rule: Similar to successor rule, except that the partition is added when a block has two or more predecessors ("backward edges $P \to Pi$"). For distinction backward edges are shown as "dashed" edges in the graph.
- Completion Rule: When $P \to P1 \mid P2$ is partition and P is not reachable from the root, then the graph in incomplete. If P1 and P2 are reachable from the root, there is a lowest common ancestor of P1 and P2, lca(P1, P2). Since P is reachable from lca(P1,P2), a partition lca(P1, P2) = $P \mid Q1 \mid ... \mid Qk$ can be created, where the union of the Qj, j=1,…,k, is the relative complement of P with respect to lca(P1,P2).

Intuitively the PPG can always be completed. For simplicity our running example does not require the completion rule. But an example for a PPG that does will be given later (see Figure 38). PQS queries are based on the complete PPG and on the interpretation of predicates as sets: each predicate represents an execution set, which is a set of execution traces for which it is true. The execution traces in if-converted code correspond to paths in the original control flow graph. The interpretation of a predicate as a set makes available set relations like subset, intersection etc. for predicates. The implementation of PQS is based on the set interpretation (see Appendix 12.3).

There are two preparation steps before the partition graph is built: first, the control flow graph is completed. Completion is necessary for the uniqueness of the predicate partitions and preciseness of disjointness. Completion requires one pass over the control flow graph and inserts empty basic blocks on critical edges. A critical edge is defined as follows: If basic block B1 has two or more successors and basic block B2 has two or more predecessors, then the edge B1 → B2 is critical. The inserted basic block is referred to as JS (“Join-Split”) block. Second, a block predicate is assigned to each basic block. For this, the compiler uses the RK algorithm. The characteristic of the RK algorithm is that it assigns the same block predicate to a set of control-equivalent basic blocks. Informally two basic blocks B1 and B2 are control-equivalent when B1 executes whenever B2 executes and vice versa. Using a single predicate for a class of control-equivalent blocks results in a more compact representation of the PPG of the predicate relations derived from the control flow graph.

We use a rolling example, which is a more elaborate version of the example in Johnson and Schlansker [40], to illustrate the predicate partition graph and PQS. Figure 29 shows the control flow graph and source code snippets. Associated with each basic block B is a block predicate P. Control-equivalent basic blocks are assigned the same block predicates. For example, B2 and B7 are control-equivalent. So are B1 and B8. The edge from B3 to B8 is critical, and the completion phase inserted JS block Bx on the edge. A JS block splits a critical edge. The acyclic predicate partition graph corresponding to the control flow graph is in Figure 30. Partitions (edges “1”),   (edges “5”) and (edges “3”) are forward partitions, partition (edges “4”) is a backward partition, and partition (edges “2”) is both, a forward and a backward partition. The corresponding if-converted code is in Figure 39, which will also discuss live ranges under PQS.

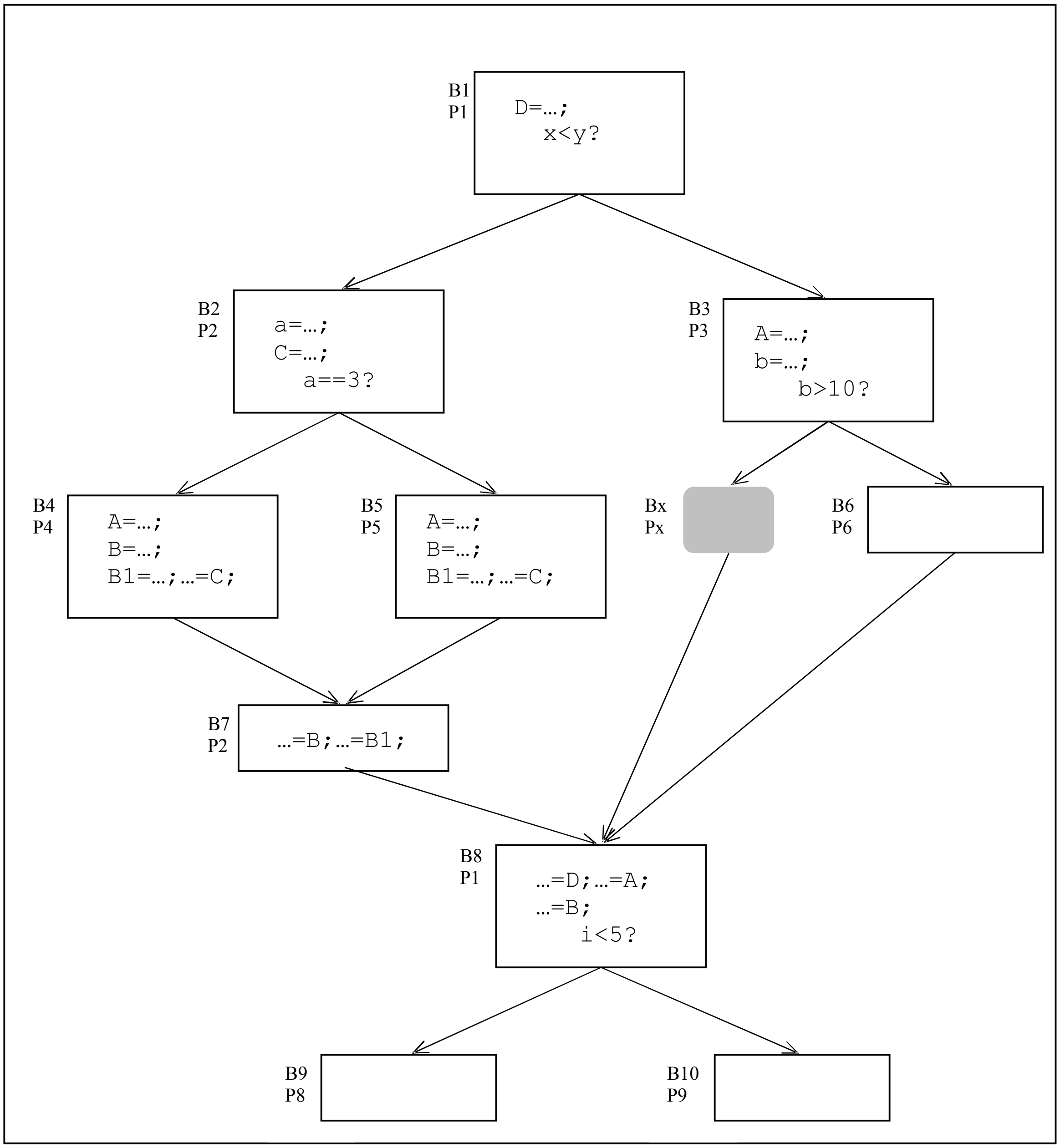
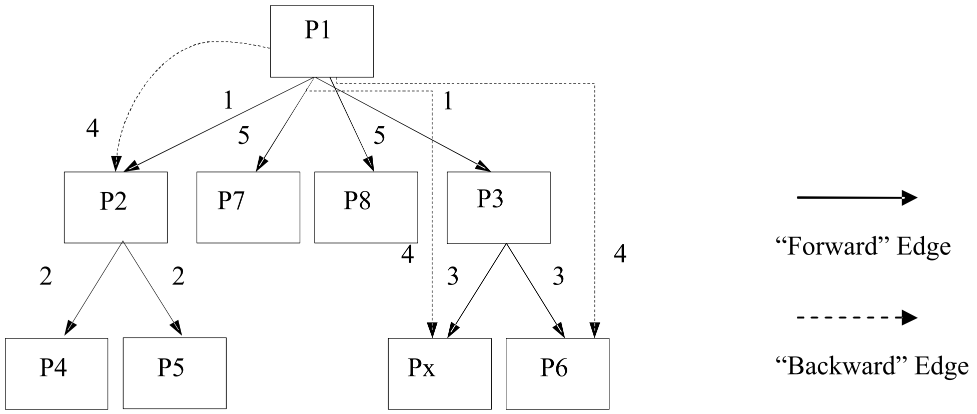


Figure 29 Example with Control flow Graph and Source Code Snippets



Root: $P1$

Forward Partitions: $P1 \to P2 \mid P3$ (“1”) , $P2 \to P4 \mid P5$ (“2”), $P3 \to Px \mid P6$ (“3”), $P1 \to P7 \mid P8$ (“5”)

Backward Partitions: $P1 \to P2 \mid Px \mid P6$ (“4”)

Completed Partitions: none.

Figure 30 Predicate Partition Graph (PPG) for Example in Figure 29

Both live variable analysis and interference calculation use PQS queries that walk the predicate partition graph (PPG) to compute accurate liveness information at each instruction. Figure 31 shows four predicated live ranges A, B, B1 and C and their predicated sets during a backward traversal of the instructions 1-20 in the if-converted code fragment of the rolling example. The figure shows the predicate set at the entry of each instruction for each variable. The predicate set is derived from the predicate set at the exit of the instruction and the qualifying predicate of the instruction. The algorithms PQS uses to find the predicate sets are in Figure 32. It specifies the “add” and “delete” interfaces the general predicate-aware interference build routine uses (see Algorithm 2). The PQS queries that traverse the PPG to find the predicate sets for candidates are listed in Appendix 12.3.

Predicated Code		Predicate Sets For Variables			
		A	B	B1	C
1:	`(P1) D=`	{}	{P3}	{}	{}
2:	`(P1) cmp P2,P3=(x<y)`	{}	{P3}	{}	{}
3:	`(P2) a=…`	{}	{P3}	{}	{}
4:	`(P2) C=…`	{}	{P3}	{}	{}
5:	`(P2) cmp P4,P5=(a==3)`	{}	{P3}	{}	{P4, P5}
6:	`(P3) A=…`	{}	{P3}	{}	{P4, P5}
7:	`(P3) b=…`	{P3}	{P3}	{}	{P4, P5}
8:	`(P3) cmp P6,p0=(b>10)`	{P3}	{P3}	{}	{P4, P5}
9:	`(P4) A=…`	{P3}	{P3}	{}	{P4, P5}
10:	`(P4) B=…; (P4) B1=…;`	{P4, P3}	{P3}	{}	{P4, P5}
11:	`(P4) …=C`	{P4, P3}	{P4, P3}	{P4}	{P4, P5}
12:	`(P5) A=…`	{P4,P3}	{P4, P3}	{P4}	{P5}
13:	`(P5) B=…; (P5) B1=…;`	{P1}	{P4, P3}	{P4}	{P5}
14:	`(P5) …=C`	{P1}	{P1, P2}	{P2}	{P5}
15:	`(P2) …=B; (P2)…=B1;`	{P1}	{P1, P2}	{P2}	{}
16:	`(P6) …`	{P1}	{P1}	{}	{}
17:	`(P1) …=D`	{P1}	{P1}	{}	{}
18:	`(P1) …=A`	{P1}	{P1}	{}	{}
19:	`(P1) …=B`	{}	{P1}	{}	{}
20:	`(P1) cmp P7,P8=(i!=5)`	{}	{}	{}	{}

- Variable A is a "partition" live range. At instruction 12 A is live under P1 and defined under P5. Since P1 = P2|P4=P3|P4|P5, A is live under P3 and P4 at the beginning of instruction 12. After (reading backwards) the definition of A under P4 in instruction 9, A is live under P3. Finally, instruction 6 is the start of the live range of A.
- Variable B is a "partition" live range. At the definition under P5 in instruction 13, it is live under P1 and P2. Since P1=P3|P4|P5 and P2=P4|P5, B is live under P3 and P4 at the beginning of instruction 13. Since there is no definition under P3, B is live at then entry of the predicated region under P3.
- Variable B1 is a "partition" live range similar to A or B.
- Variable C is a "dominate" live range. The definition under P2 in instruction 4 is the start of the live range since P2=P4|P5, so P2 dominates P4 and P5).

Figure 31 Predicated Live Ranges under PQS

PQS is powerful, but has costs. First, it requires the construction of the predicate partition graph, which—although it is linear in space and time in the order of the number of basic blocks—consumes extra memory. Second, unlike classical live variable analysis, which operates on basic blocks, PQS-based predicate live variable dataflow operates on instructions, which requires customized dataflow routines. Finally, PQS queries get invoked at every predicated instruction during the backward traversal of interference graph construction (Algorithm 2). These compile time, implementation and maintenance costs motivate the search for alternatives.

```
procedure mark_live(Set P’)
    foreach p’ ∈ P’
        if (p’ ∈ B)
            live[p’]+=x;
        else
            foreach pb ∈ B
                if (! IsDisjoint(p’,pb))
                   live[pb]+=x;
                fi;
            endfor
        fi
    endfor
endproc

procedure add(Predicate qp, Var x)
    Pset P, P’;
    // Collect all predicates p in Basis B such that
    // x is live under p.
    P = { p∈B | x live under p};
    if (qp ∈ P) return; // x live under qp already.
    P’ = lub_sum(qp, P);
    mark_live(P’);
endproc
procedure delete(Var x, Predicate qp)
    Pset P, P’
    // Collect all predicates p in Basis B such that
    // x is live under p.
    P = { p∈B | x live under p};
    if (qp ∈ P) live[qp]-=x; // x dead under qp.
    P’ = lub_diff(qp, P);
    // kill x under all predicates in B
    foreach pb in B
        live[pb]-=x;
    endfor
    mark_live(P’)
endproc
```

Figure 32 Add and Delete Routines in PQS-based Allocator

6.3 A Family of Predicate-Aware Register Allocators

This section assumes all predicate code is compiler generated. We propose predicate-aware allocation schemes based solely on classical techniques. This is based on the observation that computing partitions based on PQS at every instruction during interference graph construction and live variable analysis is not necessary for all live ranges. Specifically, PQS partitions are not necessary when either the qualifying predicates for the definitions and uses of a live range match or a definition dominates all uses. "Dominance" is derived from the control flow graph. Since each instruction has a predicate and the predicate is a block predicated, predicate dominance is control flow dominance. For other live ranges, partitions can be pre-computed at each use on demand. Building and repeatedly querying the PPG is not necessary.

There are four fundamental relations between predicated definitions and uses (Figure 33). Predicated live ranges are classified based on the original control-flow region the predicated code is derived from. It is important to keep the correspondence between blocks and qualifying predicates in mind. A definition is clearly the start of the live range when the qualifying predicates of the definition and use match. This is also the case when the qualifying predicate of the definition dominates the qualifying predicates of the uses. When multiple definitions reach a use, two cases are possible. First, when definitions form a partition, the qualifying predicates of the definitions are mutually disjoint and the first definition in the hyperblock is the start of the live range. In this case the allocator gets precise disjointness by tracking liveness under all predicates that reach the use. Therefore it can track liveness under all definition predicates reaching a use rather than the qualifying predicate of the use (instruction). For example, in the partition case in Figure 33, instead of tracking liveness under P3, recording liveness under P1 and P2 would give precise disjointness information. In this scenario, the definition of V under P2 (or P1) would kill liveness under P2 (or P1). Any subsequent–in the backward traversal–variables (defined or used) under P2 (or P1) do not interfere with V. This would not be the case if the live range were tracked using P3, unless a system like PQS partitioned P3 at the definition of V qualified under P2. Second, when definitions don't form a partition ("overlap"), recording liveness under the reaching predicates would find the start of the live range, but disjointness would be conservative. For example, variables under qualifying predicate P2 could interfere with V, although V might have been killed under P2, since V would be live under P1, too (see "overlap" in Figure 33).

The live range for a variable defined in the region *and* live is completed (=made strict relative to the region) by adding pseudo definitions into region blocks based on two rules: first, if the variable V is live at entry of two successors, follow both paths. Second, if block B1 has two successors, B2 and B3, and variable V is live at entry in B2, but dead at entry in B3, insert a pseudo definition at the beginning of B2. The pseudo definition does not start the live range, but splits it into separate components. We note that the predicated live ranges can be classified based on the original control flow graph.

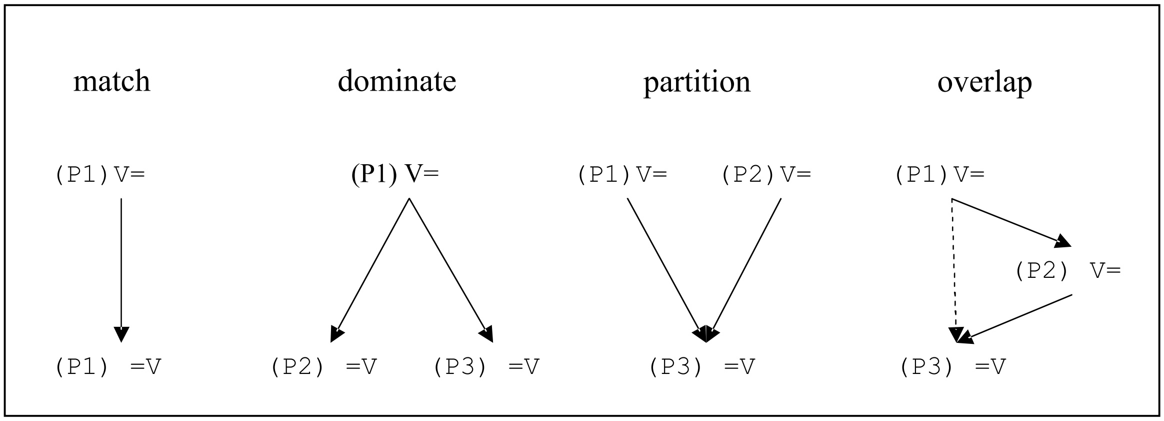


Figure 33 Fundamental Relations between Predicated Definitions and Uses

Our rolling example (Figure 30) illustrates the four fundamental relations. Live ranges D, a, and b are defined and used under a single predicate ("match"). In live range C the definition under (P2) dominates the uses under (P4) and (P5) ("dominate"). In live range A the qualifying predicates of the definitions (P3, P4 and P5) form a partition for the use under (P1) ("partition"). Live range B has uses under (P1) and (P2). The qualifying predicates (P4) and (P5) form a partition for (P2) ("partition"). For (P1) there is no partition. Since B is live at the entry of the predicated region, there is a pseudo definition of B in block 3 under (P3), since B is live at entry in block 3, but dead at entry in block 2.

The start of predicated live ranges can be found performing live variable analysis before if-conversion. After a region is if-converted, first definitions can be marked in a forward sweep over all instructions in the (linear) if-converted region, starting at the first instruction in the region entry block with the live-at-entry vector and recording definitions: if a variable is defined under a predicate, the variable is not live-at-entry and no other definition of the variable has been seen, this must be the first definition.

Based on the types of predicate live ranges, three strategies for predicate-aware register allocation can be defined that model predicate live ranges with increasing accuracy:

Strategy 1: Dominate-or-Match

The qualifying predicates of instructions that use a variable form the predicate set for the variables. For live ranges with matching qualifying predicates for definition and uses, interference is precise. This is true also when the definition predicate dominates all use predicates.

Strategy 2: Partition Tracking

In addition to Strategy 1, live ranges are recorded under qualifying predicates of the (possibly pseudo) definitions that reach a use, if this set is a partition and the qualifying predicate of the use post-dominates each definition predicate. The qualifying predicates at the definitions are either in the partition or -in case the predicate is from a pseudo definition- dominate a partition predicate.

Figure 34 has the predicated code from our example and considers two live ranges A and B to illustrate strategy 2. For live range A, {P3, P4, P5} reach the use under (P1). Since P3, P4 and P5 are mutually disjoint and the use post-dominates the definitions, this partition is the predicate set at the use of A. Live range B is similar to live range A, except that B is completed by a (implicit) pseudo definition at the entry of block 3. Completion ensures that all live ranges with uses in the region are strict and enables partition formation at uses. A live range is strict when there is a definition on every path to a use.

After if-conversion each read operand (“use) is augmented with a list of qualifying predicates that represent the qualifying predicates of its reaching definitions. Strategy 2 relies on reaching definition analysis per predicated region. When more than one definition predicate reaches a use and the predicates are disjoint, record the partition that represents reaching qualifying predicates at each use. In this case, the original (qualifying) predicates at the definitions are either in the partition or dominate a partition predicate.

	Predicated Code	Predicate Sets For Variables			
		A	B	B1	C
1:	(P1) D=	{}	{P3}	{}	{}
2:	(P1) cmp P2,P3=(x<y)	{}	{P3}	{}	{}
3:	(P2) a=…	{}	{P3}	{}	{}
4:	(P2) C=…	{}	{P3}	{}	{}
5:	(P2) cmp P4,P5=(a==3)	{}	{P3}	{}	{P4, P5}
6:	(P3) A=…	{}	{P3}	{}	{P4, P5}
7:	(P3) b=…	{P3}	{P3}	{}	{P4, P5}
8:	(P3) cmp P6,p0=(b>10)	{P3}	{P3}	{}	{P4, P5}
9:	(P4) A=…	{P3}	{P3}	{}	{P4, P5}
10:	(P4) B=…; (P4) B1=…;	{P3,P4}	{P3}	{}	{P4, P5}
11:	(P4) …=C	{P3,P4}	{P3,P4}	{P4}	{P4, P5}
12:	(P5) A=…	{P3,P4}	{P3,P4}	{P4}	{P5}
13:	(P5) B=…; (P5) B1=…;	{P3,P4,P5}	{P3,P4}	{P4}	{P5}
14:	(P5) …=C	{P3,P4,P5}	{P3,P4,P5}	{P4,P5}	{P5}
15:	(P2) …=B; (P2)…=B1;	{P3,P4,P5}	{P3,P4,P5}	{P4,P5}	{}
16:	(P6) …	{P3,P4,P5}	{P3,P4,P5}	{}	{}
17:	(P1) …=D	{P3,P4,P5}	{P3,P4,P5}	{}	{}
18:	(P1) …=A	{P3,P4,P5}	{P3,P4,P5}	{}	{}
19:	(P1) …=B	{}	{P3,P4,P5}	{}	{}
20:	(P1) cmp P7,P8=(i!=5)	{}	{}	{}	{}

- Variable A is a “partition” live range. The use of A in instruction 18 is reached by definition of P3, P4 and P5, which form a partition of the qualifying predicate P1. Liveness of A is tracked under P3, P4 and P5.
- Variables B and B1 are “partition” live ranges similar to A. The live range of B is “completed” with a pseudo-definition in Block B3. P3, B is live at then entry of the predicated region under P3.
- Variable C is a “dominate” live range. The definition under P2 in instruction 4 is the start of the live range since P2=P4|P5 (so P2 dominates P4 and P5).

Figure 34 Predicate-aware Allocation with Partitions

The following theorem lists the live ranges whose interferences can be modeled precisely by strategy 2.

Theorem 6-1 (Characterization of Simple Live Range Tracking)

Strategy 2 can model interferences precisely for the following live ranges:

- Definition and use predicate match
- Definition predicate dominates use predicate

- Definition predicates form a partition. Use predicate post-dominates all partition predicates.
- Two definition predicates reaching a use are on at most one execution trace (or execution path in the original control flow region).

Proof:

Precise interference means for each predicated live range L:

When L is recognized as live at a given program point, L is actually live.

When L is recognized as dead, it is actually dead.

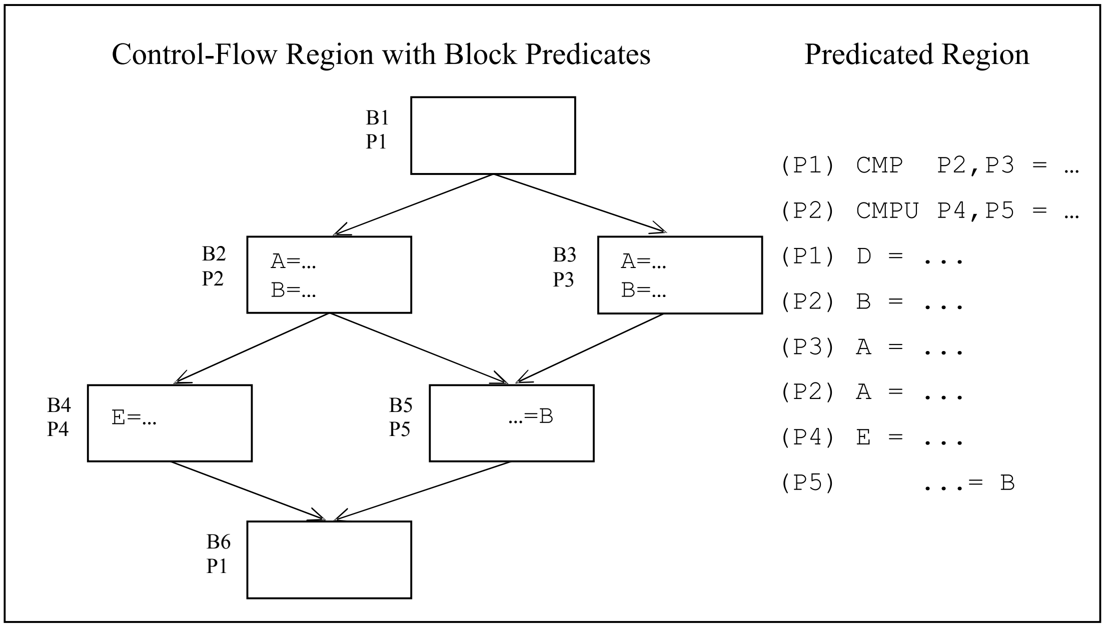
When L is recognized as disjoint from another live range L’, it is actually disjoint.

When L is recognized as interfering with L’, it is actually interfering.

The theorem is clear for the simple cases, match and dominate. In this case the predicate set of a live range consists of the qualifying predicates seen at its uses. The matching or dominating definition stops the live range (strategy 1). For the remaining cases we need to develop some intuition first. Tracking a live range under the qualifying predicate of the use ensures that in the predicated region disjointness is precise with respect to instructions that are not on a path to the use in the original control flow graph. In case the definition predicates form a partition and the use predicate post-dominates all partition predicates, then all paths starting at definitions end at the use. There cannot be an off path instruction in the predicated region that would introduce an interference that is not visible in the original control flow graph. Therefore tracking the live range under partition predicates cannot introduce new interferences. The case of two definitions reaching a use but the definition predicates don’t form a partition can be reduced to the partition case. Since there is only one path that contains both definitions, there must exist a split block dominated by the first definition. Inserting a pseudo-definition in the successor of the split block that is not on the path to the second definition ensures the partition property: since the qualifying predicate of the pseudo def is disjoint from the qualifying predicate of the second definition, they form a partition at the use. In this case one partition predicate (from the pseudo definition) will not be a definition predicate, but dominated by it (the definition predicate of the first definition). This proves the theorem for two definitions. The general case of N definitions is similar to this special case. □

Consider the example code in Figure 35 to visualize the difference between use and partition tracking, when the use does not post-dominate the definition. P2|P3 form a partition for live range B, but the use under P5 does not post-dominate P2. In the predicated code there could be an off-path instruction like the definition of E under P4 “before” the use of B under P5. If liveness of B were tracked under partition predicates P2|P3, E and B would interfere, since P2 and P4 are not disjoint. On the other hand, if liveness of B is tracked under P5, E and B cannot interfere, since clearly P4 and P5 are disjoint. This scenario cannot happen when the use post-dominates all partition predicates, since there cannot be an “off-path” instruction on the execution trace.

The remaining live ranges require a more sophisticated method to model interferences precisely. There are two cases left: First, the use does not post-dominate the partition predicates. Second, two or more definitions overlap on more than one execution trace (or execution path in the original control flow graph). The first case can be handled by tracking the live range under the use and the partition predicates. The second case is reduced to partition, dominate or match live ranges by splitting. Splitting is described below.



```
(P1) CMP  P2,P3 = …
(P2) CMPU P4,P5 = …
(P1) D = ...
(P2) B = ...
(P3) A = ...
(P2) A = ...
(P4) E = ...
(P5)     ...= B
```

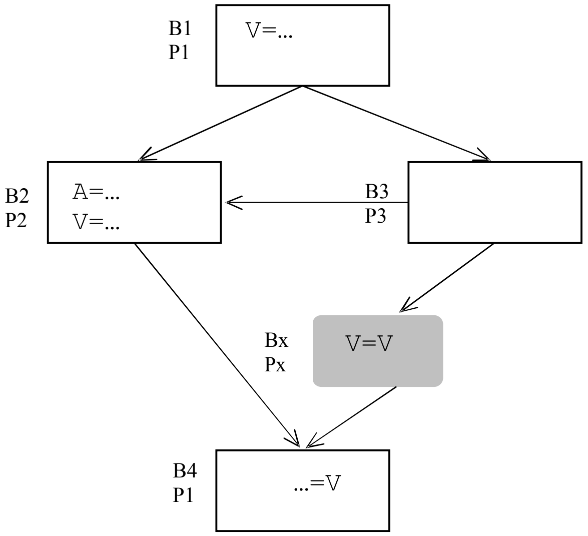
Figure 35 Extra Interference with Partition Tracking

Strategy 3: Use-and-partition Tracking

In addition to strategy 2, track live variables under use-and-partition predicates and "split" live ranges when two definitions overlap on more than one path.

When the use does not post-dominate the definition, the use predicate (=qualifying predicate of the instruction containing the use) gets associated with the partition predicates. This is necessary for precise disjointness information: when the use does not post-dominate all predicates in a partition (of two or more predicates), disjointness could be conservative. Therefore, the live range is tracked under the qualifying predicate of the use *and* the partition predicates. Since the partition predicates represent disjoint portions of execution traces, precise disjointness is due to the following rule used during interference calculation: at any given instruction, if the qualifying predicate of a definition of live range L1 interferes with the use predicate of live range L2, but *not* with any of its associated partition predicates, then live ranges L1 and L2 are (actually) disjoint at this point. This gives precise disjointness: First, when the use does not post-dominate the definitions, there can be instructions on the execution trace that are not on any path from the definition to the use. Since the qualifying predicate of the use is disjoint from the qualifying predicate of these instructions, no imprecise interference can be encountered. Second, false interferences could be recorded with variables in instructions on paths to a definition, but the rule above is preventing this, since at every definition the qualifying predicate is removed from the partition. This argument has been used in the proof for Theorem 6-1 also.

In case of definition overlap on more than one execution path, additional live range splitting is necessary. This is achieved by inserting an identity move under the qualifying predicate of the definition. This move only changes predicate tracking for the live range.



- On paths B1-B2 and B1-B3-B2 definitions of V overlap.
- In this case the live range for V cannot be transformed into a partition live range.
- The pseudo move in JS block Bx splits the live range of V into a “partition” and a “dominate” live range.

Figure 36 Complex Live Range Tracking

Figure 36 illustrates a live range V in a control flow graph snippet. The definitions for V overlap on more than one path to the use. There are two definitions of V in blocks B1 and B2. The use in block B4 post-dominates the definitions, but the definitions in blocks B1 and B2 overlap on paths $1 \to 2$ and $1 \to 3 \to 2$. Tracking the live range of V under the reaching predicates P2|P1 would give extra interferences, since P1 interferes with P2. The trick is adding an identical move in block Bx, which is inserted by control flow graph completion. The use in block B4 is recorded under P2 and Px, which form a partition. Since P1 dominates Px, the original live range has been split into a “simple” live range (handled by strategy 2) and a “dominate” live range. In general, splitting can also result in a partition live range, where the use does not post-dominate the definitions. We note that this kind of live range splitting can be achieved with SSA representation [59]) in the original control flow region. When translating out of SSA we would get the moves that split a complex live range.

From the discussion it is clear that strategy 3 models interference precisely when a live range has two definitions that overlap on one execution path. The identity move can be inserted where the two definitions merge. The general case is

Theorem 6-2 (Characterization of Complex Live Range Tracking)

Strategy 3 can model interferences precisely for the following live ranges:

- Use predicate does not post-dominate partition predicates
- When definitions overlap on more than one path, the live range can be split and handled by strategy 2 or the case above.

Proof:

Preciseness for the first case is clear: The qualifying predicate of the use avoids interferences with an off-path instruction, which could be on the trace from a definition to a use. Partition tracking asserts there is no conservative interference with an instruction on the path from the entry code to the definition. The interference rule is modified: if, at a given instruction in the interference graph construction, a qualifying predicate interferes with the use predicate from a live candidate, but not with the associated partition predicates, the live range under the qualifying predicate is disjoint from the live candidate.

Overlapping live ranges on multiple paths can be split into simpler live ranges: Assume the live range has n> 2 definitions. Like in Figure 36 an identical move can be inserted at a merge point of any two definitions. The live range section with the two definitions and the use in the identical "mov" is either a partition live range or can be modeled by use-and-partition tracking. This splitting technique can be applied iteratively until a split live range has only two definitions. This proves the theorem since it holds in case n=2.

□

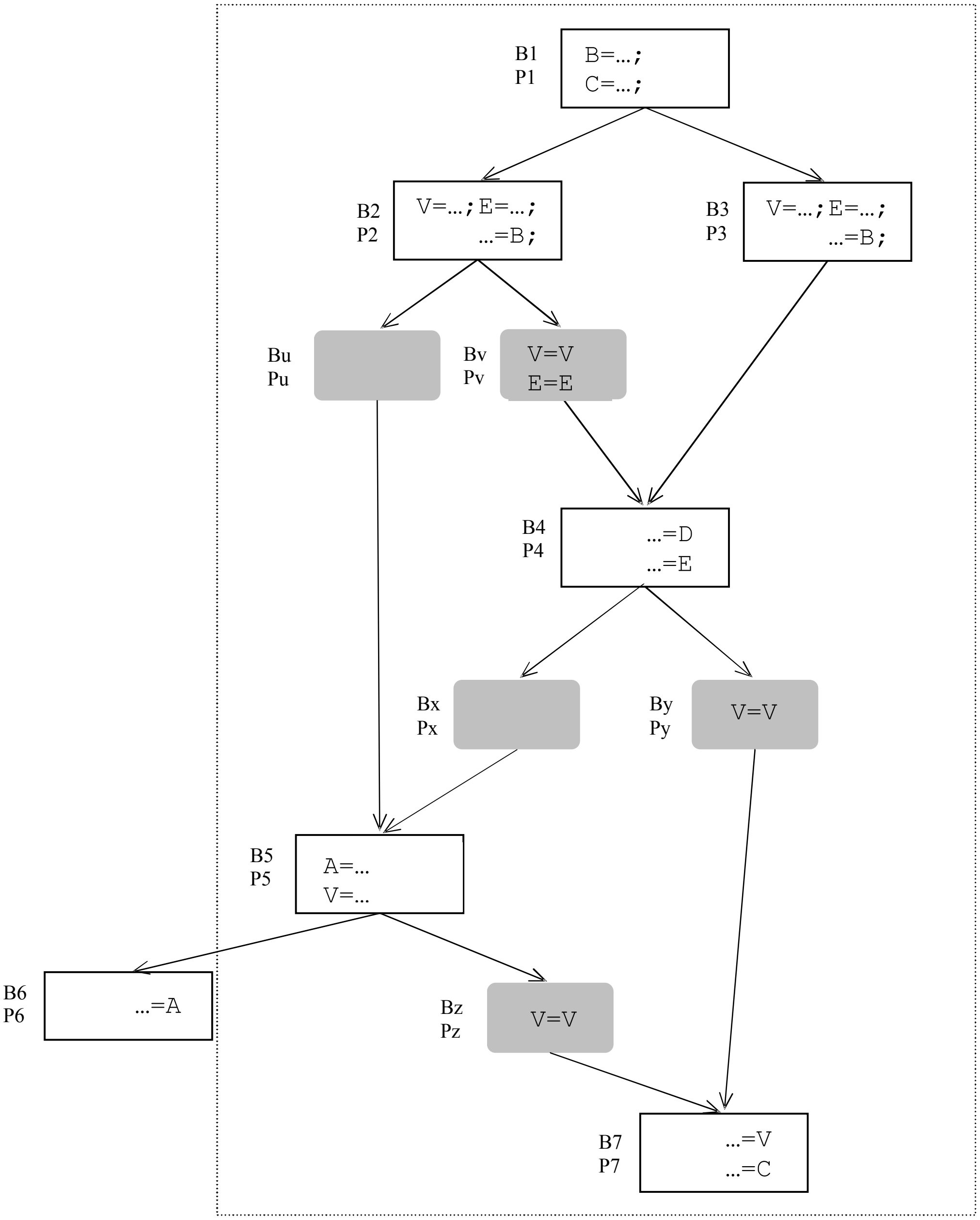


Figure 37 Completed Candidate Region and Live Ranges

We identified four types on predicated live ranges – match, dominate, partition and overlap –and two methods for interference modeling– "simple" and "complex", and three implementation strategies. The simple method models interferences of all "match", "dominate" and some "partition" live ranges precisely, and remaining live ranges conservatively. The complex method models all live range precisely. In implementation strategy 1, only match and dominate live ranges are modeled precisely. Strategy 2 models all simple live ranges, which include all "match", "dominate" and some "partition" live ranges, precisely, and strategy 3 models interferences of all live range precisely. Strategy 3 is equivalent to a PQS based implementation when predicated code is derived form control-flow code. The rest of the section discusses a detailed example. Figure 37 shows the snippet of a completed control flow graph with block predicates. Bu, Bv, Bx, By and Bz are JS blocks inserted during completion. The candidate region for if-conversion includes all blocks but B7, which is an exit block. Block 6 is the merge point for the region. Shown are also six live ranges A, B, C, D, E and V. A is defined in the region, but used outside. B and C are examples for "dominate" live range, D is used in the region and escapes at the entry B1, E is a "partition" live range and V an "overlap" live range, which has multiple definitions on a path that can reach a use.

Figure 39 shows the core predicated code ("hyperblock") corresponding to the superblock region in Figure 37. For the six live ranges their predicate sets are shown as they are recorded in a backward traversal of the hyperblock. The definition of live range A under predicate P4 dominates the exit at which A is live (Line 16, B7). Thus A is tracked under P4 when it enters the predicated region. For live range B, the definition predicate P1 dominates the use predicates P2 and P3. The same holds for live range C, where P1 dominates the use predicate P6. Live range D has no definition. At the entry of the region D becomes live under P0, the true predicate. E is an example for a partition live range, but the use predicate P5 does not post-dominate the definition predicate P2. In this case, a partition can be formed because the control flow graph has been completed before if-conversion. The partition for E is (Pv, P3), where P2 dominates Pv and P5 post-dominates all partition predicates.

Root:	$P1$
Forward Partitions:	$P1 \to P2 \mid P3, P2 \to Pu \mid Pv, P4 \to Px \mid Py, P5 \to P6 \mid Pz$
Backward Partitions:	$P4 \to P3 \mid Pv, P5 \to Pz \mid Px, P7 \to Pz \mid Py$
Completed Partitions:	$P1 \to P4 \mid Pu, P1 \to P5 \mid Py, P1 \to P7 \mid P6$

Figure 38 Predicate Partition Graph for Figure 37

Predicated Code	Predicate Sets For Variables					
	A	B	C	D	E	V
1: `(P1) B=`	{}	{}	{}	{P4}	{}	{}
2: `(P1) C=`	{}	{P2, P3}	{P7}	{P4}	{}	{}
3: `(P1) cmpu P7,P0=...`	{}	{P2, P3}	{P7}	{P4}	{}	{}
4: `(P1) cmpu P4,P5=...`	{}	{P2, P3}	{P7}	{P4}	{}	{}
5: `(P1) cmpu P2,P3=...`	{}	{P2, P3}	{P7}	{P4}	{}	{}
6: `(P2) V=`	{}	{P2, P3}	{P7}	{P4}	{}	{}
7: `(P3)     =B`	{}	{P2, P3}	{P7}	{P4}	{}	{Pz(Pv)}
8: `(P2) E=`	{}	{P2}	{P7}	{P4}	{}	{Pz(Pv)}
9: `(P3) E=`	{}	{P2}	{P7}	{P4}	{Pv}	{Pz(Pv)}
10: `(P3) V=`	{}	{P2}	{P7}	{P4}	{Pv,P3}	{Pz(Pv)}
11: `(P2)     =B`	{}	{P2}	{P7}	{P4}	{Pv,P3}	{Pz(Pv,P3)}
12: `(P4) cmpu Px,Py=...`	{}	{}	{P7}	{P4}	{Pv,P3}	{Pz(Pv,P3)}
13: `(P5) V=`	{}	{}	{P7}	{P4}	{Pv,P3}	{Pz(Pv,P3)}
14: `(P5) cmpu P6,Pz=...`	{}	{}	{P7}	{P4}	{Pv,P3}	{Py, Pz(Pv,P3)}
15: `(P5) A=`	{}	{}	{P7}	{P4}	{Pv,P3}	{Py, Pz(Pv,P3)}
16: `(P4)     =E`	{P5}	{}	{P7}	{P4}	{Pv,P3}	{Py, Pz(Pv,P3)}
17: `(P4)     =D`	{P5}	{}	{P7}	{P4}	{}	{Py, Pz(Pv,P3)}
18: `(P5) br .b6`	{P5}	{}	{P7}	{}	{}	{Py, Pz(Pv,P3)}
19: `(Pz) V=V`	{}	{}	{P7}	{}	{}	{Py, Pz(Pv,P3)}
20: `(P7)     =C`	{}	{}	{P7}	{}	{}	{Py, Pz}
21: `(P7)     =V`	{}	{}	{}	{}	{}	{Py, Pz}

- A is a “dominate” live range. It is used in B6 under predicate P6. P5 (or B5) dominates P6 (or B6) and the definition under P5 is the start of the live range for A.
- B is a “dominate” live range with one definition and two uses.
- C is a “dominate” live range with one definition and one use.
- D has no definition inside the hyperblock. It is tracked under the qp of its use.
- E is a “partition” live range. The use does not post-dominate its definitions. E is partitioned into a “dominate” and a “partition” live range, where the use in block B4 post-dominates the definition in block B3 and the definition in the pseudo move E=E in block Bv. Liveness for the partitioned live range is tracked under partition predicate Pv and P3. Since P2 dominates Pv, the start of the live range for E is recognized at instruction 8.
- V is a “complex” live range, since the definition in B2 or B3 can overlap with the definition in B5 on a path to the use in B7. The live range of V is split into two “dominate”, one “partition” and one “complex” (with two definition) live range: The two “dominate” live ranges stretch from B2-Bv and B5-Bz respectively, the partition live range is Bz-By-B7, and the complex live range with two definitions is Bv-B3-By. In line 21 the two definitions (from pseudo moves V=V) in By and Bz reach the use of V under P7. Thus liveness of V is tracker under {Py, Pz}. In instruction 19, liveness under Pz is killed. The use in the same (pseudo move) instruction is the end of the “complex” (sub-) live range of V. Since the use does not post-dominate its reaching definitions, liveness is tracked under the use predicate Pz, and the reaching definitions predicates Pv, P3. Instruction 13 kills liveness under Py, since P5 dominates Py. Instruction 6 kills liveness under Pv, since P2 dominates Pv. Since Pv is the “last” predicate under which V is live, instruction 6 is the start of the live range.

Figure 39 Predicate Sets for Live Ranges Figure 37 with Complex Tracking

Predicated Code	Predicate Sets For Variables					
	A	B	C	D	E	V
1: `(P1) B=`	{}	{}	{P7}	{P4}	{}	{}
2: `(P1) C=`	{}	{P2, P3}	{P7}	{P4}	{}	{}
3: `(P1) cmpu P7,P0=...`	{}	{P2, P3}	{P7}	{P4}	{}	{}
4: `(P1) cmpu P4,P5=...`	{}	{P2, P3}	{P7}	{P4}	{}	{}
5: `(P1) cmpu P2,P3=...`	{}	{P2, P3}	{P7}	{P4}	{}	{}
6: `(P2) V=`	{}	{P2, P3}	{P7}	{P4}	{}	{}
7: `(P3)     =B`	{}	{P2, P3}	{P7}	{P4}	{}	{Pv}
8: `(P2) E=`	{}	{P2}	{P7}	{P4}	{}	{Pv}
9: `(P3) E=`	{}	{P2}	{P7}	{P4}	{Pv}	{Pv}
10: `(P3) V=`	{}	{P2}	{P7}	{P4}	{P4}	{Pv}
11: `(P2)     =B`	{}	{P2}	{P7}	{P4}	{P4}	{Py}
12: `(P4) cmpu Px,Py=...`	{}	{}	{P7}	{P4}	{P4}	{Py}
13: `(P5) V=`	{}	{}	{P7}	{P4}	{P4}	{Py}
14: `(P5) cmpu P6,P7=...`	{}	{}	{P7}	{P4}	{P4}	{P7}
15: `(P5) A=`	{}	{}	{P7}	{P4}	{P4}	{P7}
16: `(P4)     =E`	{P6}	{}	{P7}	{P4}	{P4}	{P7}
17: `(P4)     =D`	{P6}	{}	{P7}	{P4}	{}	{P7}
18: `(P5) br .b6`	{P6}	{}	{P7}	{}	{}	{P7}
19: `(P7)     =C`	{}	{}	{P7}	{}	{}	{P7}
20: `(P7)     =V`	{}	{}	{}	{}	{}	{P7}

- A is a "dominate" live range. It is used in B6 under predicate P6. P5 (or B5) dominates P6 (or B6) and the definition under P5 is the start of the live range for A.
- B is a "dominate" live range with one definition and two uses.
- C is a "dominate" live range with one definition and one use.
- D has no definition inside the hyperblock. It is tracked under the qp of its use.
- E is a "partition" live range. The use does not post-dominate its definitions. At instruction 9 is live at exit under predicate P4 and defined under predicate P3. Since there is a partition P4=Pv|P3in the PPG, E becomes live at entry under Pv. In instruction 8, E is defined under P2. Since P2 dominates Pv, instruction 8 is the start of the live range of E.
- V is a "complex" live range. At instruction 13 it is defined under P5. Since Py is the relative complement to P5 and P7 (the lowest common ancestor of P5 and P7 is P1, and Py is not on any path from P1 to P5), V becomes live under Py at the entry of instruction 13. At instruction 10, V is defined under P3. Since

Figure 40 Predicate Sets for Live Ranges in Figure 37 with PQS

The live range for V is another example for a complex live range, but – unlike E – it is characterized that multiple definitions on one path reach the use in B6. In this case the live range is first split by inserting an identity move into JS block Bz into two partition live ranges. The first live range is spanned by the definition predicates P4 and Pz, and the use predicate P6. Since P6 does not post-dominate P4, the partition in Line 19 is {Py, Pz}, where P4 dominates Py and P6 post-dominates both Py and Pz. The second live range is spanned by the definition predicates P2 and P3, and the "pseudo" use in Pz. Since Pz post-dominates neither P2 nor P3, the partition records both the use predicate and the definition predicates. Only after all definitions have been seen, the live range is no longer live under Pz (Line 4).

7 Register Allocation for Speculated Code

IA-64 supports control- and data speculation to enable the compiler to speculatively hoist a load and its dependent uses across a branch (control), a store (data) or both (control and data). Compiler heuristics decide when speculation is beneficial. The compiler has to prepare for the case of an exception or fault at a speculated load. It provides for mis-speculation by generating recovery code, which may re-execute (at runtime) the speculated dependence chain non-speculatively. Re-execution starts at the original program point where the load would have been executed non-speculatively. The challenge for the register allocator with respect to control speculation is correctness. In data speculated code the allocator can reduce the interferences for the “tail” of some data speculated live ranges.

7.1 Control Speculation

Itanium provides a speculative load (ld.s) and a validating check speculation (chk.s) instruction for control speculation (*breaking the branch barrier*). An ld.s causes a deferred exception token being set in the destination register of the speculated load in case of an exception or fault, e.g. in the case of a page fault. For integer registers, the NaT (= Not a Thing) is encoded in an extra bit (“NaT bit”) for the register. For floating-point register, the deferred exception token is encoded as a special value in the register. We focus on the integer case, which—for the register allocator—is more challenging. The Itanium processor propagates NaTs to the destination register of an instruction when any source register has its NaT bit set. If a NaT bit is set for the source register of the chk.s, execution branches to recovery code, which re-executes a non-speculative instance of the speculative load and all speculated instructions of its dependence chain, and branches back to the bundle after the chk.s. Non-speculative re-execution of the speculated code at the place where the code would have been executed without speculation ensures program correctness in case of a mis-speculation. Example **7** shows an example for control speculation and two ways of generating recovery code. The two loads in lines 7 and 8 are control speculated across the branch in line 6. The “add” in line 9, which is flow dependent on the load in line 8, is speculated also. The first method for generating recovery code duplicates the dependence chain for the speculative load and all its speculated, dependent instructions. The speculated load becomes non-speculated in the recovery code. The second method shows no speculative load in the recovery code. This method saves one chk.s and code size in the recovery code sections. Note that in the optimized recovery code sequence the ld.s destination and the chk.s use don't match: the chk.s will fire when the NaT bit for V9 is set. This is the case when V9 inherits the NaT bit from V7 or V8. In other words, NaT bits may be propagated along the dependence chain originating with a speculated load ld.s.

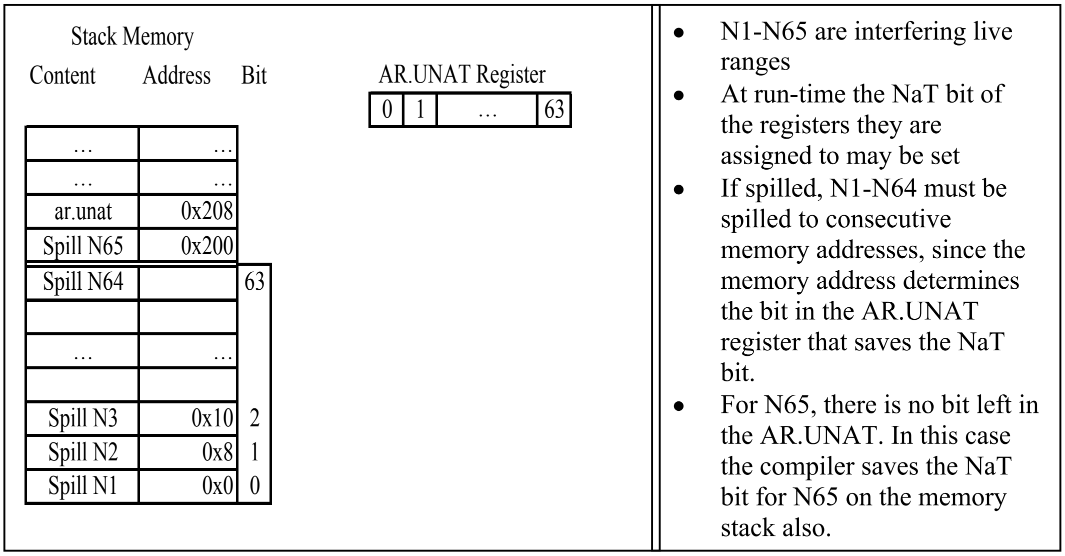
```
      a. Original Code     b. Speculated Code     c. Optimized Recovery Code

                           ld8.s V7=[V6]           ld8.s V7=[V6]
                           ld8.s V8=[V7]           ld8.s V8=[V7]
      add   V5=V4,V3       add   V5=V4,V3;;        add   V5=V4,V3;;
                           add   V9=V9,V8          add   V9=V9,V8
      br    cont           br       cont           br    cont
      ld8   V7=[V6]        chk.s V7,rec1
      ld8   V8=[V7];; r1:  chk.s V8,rec2           chk.s V9,rec1
      add   V9=V9,V8
cont: ...                  cont: ...               cont: ...
                     rec1: ld8   V7=[V6]     rec1: ld8   V7=[V6]
                           ld8.s V8=[V7];;         ld8   V8=[V7];;
                           add   V9=V9,V8          add   V9=V9,V8
                           br       r1             br    cont
                     rec2: ld8   V8=[V7];;
                           add   V9=V9,V8
                           br    cont
```

Example 7 Control Speculation and Two M ethods of Recovery Code Generation

7.1.1 NaT Propagation and Spill Code

The Itanium processor propagates NaTs to the destination register of an instruction when any source register has its NaT bit set. The compiler has to model NaT propagation of the processor to avoid NaT consumption faults. A NaT consumption fault occurs, for example, when the source register of a regular store has the NaT bit set. To avoid the NaT consumption fault, the Itanium ISA supports a special store instruction, st8.spill, which saves the NaT bit of the source register in the 64bit AR.UNAT application register and does not cause a NaT consumption fault. In Example **7** b. and c., a NaT bit may be set (at run-time) for the physical registers assigned to V7, V8 and V9: it could be set for

V7 at the speculated load in line 2 and propagate to V8 and V9. Or the NaT bit of the register assigned to V8 could be set at the speculated load in line 3 and propagate to (the assigned register of) V9 at the “add” in line 5. A set NaT bit is cleared when the speculated dependence chain that produces the NaT gets (re-)executed non-speculatively in recovery code. Modeling processor NaT propagation is crucial for register spilling to avoid possible NaT consumption faults. At a spill the register allocator must know whether or not the NaT bit of the register to be spilled could be set. The modeling can be done in a separate data flow analysis pass for speculated loads or in the code scheduler, which is responsible for generating the recovery code. As a result, the code generator marks each instruction (or even each operand per instruction) that could have a destination register with a set NaT bit.



- N1-N65 are interfering live ranges
- At run-time the NaT bit of the registers they are assigned to may be set
- If spilled, N1-N64 must be spilled to consecutive memory addresses, since the memory address determines the bit in the AR.UNAT register that saves the NaT bit.
- For N65, there is no bit left in the AR.UNAT. In this case the compiler saves the NaT bit for N65 on the memory stack also.

Figure 41 Spill Addresses and AR.UNAT Register

The rest of this section assumes that a register may have its NaT bit set. When such a register is spilled, the code generator has to use a special spill/fill instruction (st8.spill/ld8.fill), which save/restore the registers NaT bit to/from the AR.UNAT application register. The bit location in the AR.UNAT is determined by 6 significant bits (bits 8:3, see manual [13]) of the spill (=stack memory) address. To track the AR.UNAT bit vs. memory address correspondence, the register allocator has to allocate contiguous memory in the local stack frame to the spilled speculated live ranges (Figure 41). Spilled live ranges that don't interfere may share the same spill address. Spilled live ranges that interfere can neither spill to the same address nor to addresses that map to the same bit in the AR.UNAT, which is a 64-bit register and can hold 64 (NaT) bits that are live simultaneously. When more than 64 speculated live ranges are live simultaneously, the AR.UNAT would overflow. Perhaps the best way to think about spilling of a speculated live range (its assigned register may have its NaT bit set) is spilling of a pair (value, NaT). Both components of the pair can have interferences. When more than 64 speculated live ranges interfere simultaneously and have to be spilled, 64 spill addresses are not enough. This means, the AR.UNAT register would overflow and could not hold all corresponding NaT bits. In this case the allocator can use the following “naïve” spill/fill scheme for speculated live ranges (Figure 42).

Spill code sequence in case of AR.UNAT overflow

```
mov rs=ar.unat          // save ar.unat register
st8.spill [rm]=rx, 8;;  // rm = rm + 8
                        // if rx has NaT bit set,
                        // it is stored in the ar.unat
st8[rm]=rs              // save ar.unat for later fill
mov ar.unat=rs;;        // restore ar.unat register
```

Fill code sequence in case of AR.UNAT overflow

```
mov rs=ar.unat          // save ar.unat register
ld8 rh=[rm],-8;;        // rm = rm – 8.
mov ar.unat=rh          // ar.unat now contains NaT bit for
                        // candidate to be loaded in rx
ld8.fill rx=[rm]
mov ar.unat=rs;         // restore ar.unat register.
```

Figure 42 Spill/Fill Code in case of AR.UNAT overflow

When the AR.UNAT overflows the allocator saves the AR.UNAT in a general register, spills the (register of the) speculated live range, then spills the AR.UNAT and finally restores the original AR.UNAT. Filling is reversing the spills: after saving the original AR.UNAT in a general register, it loads the spilled AR.UNAT, then the spilled live range, and finally restores the original AR.UNAT. The sequences ensure that the

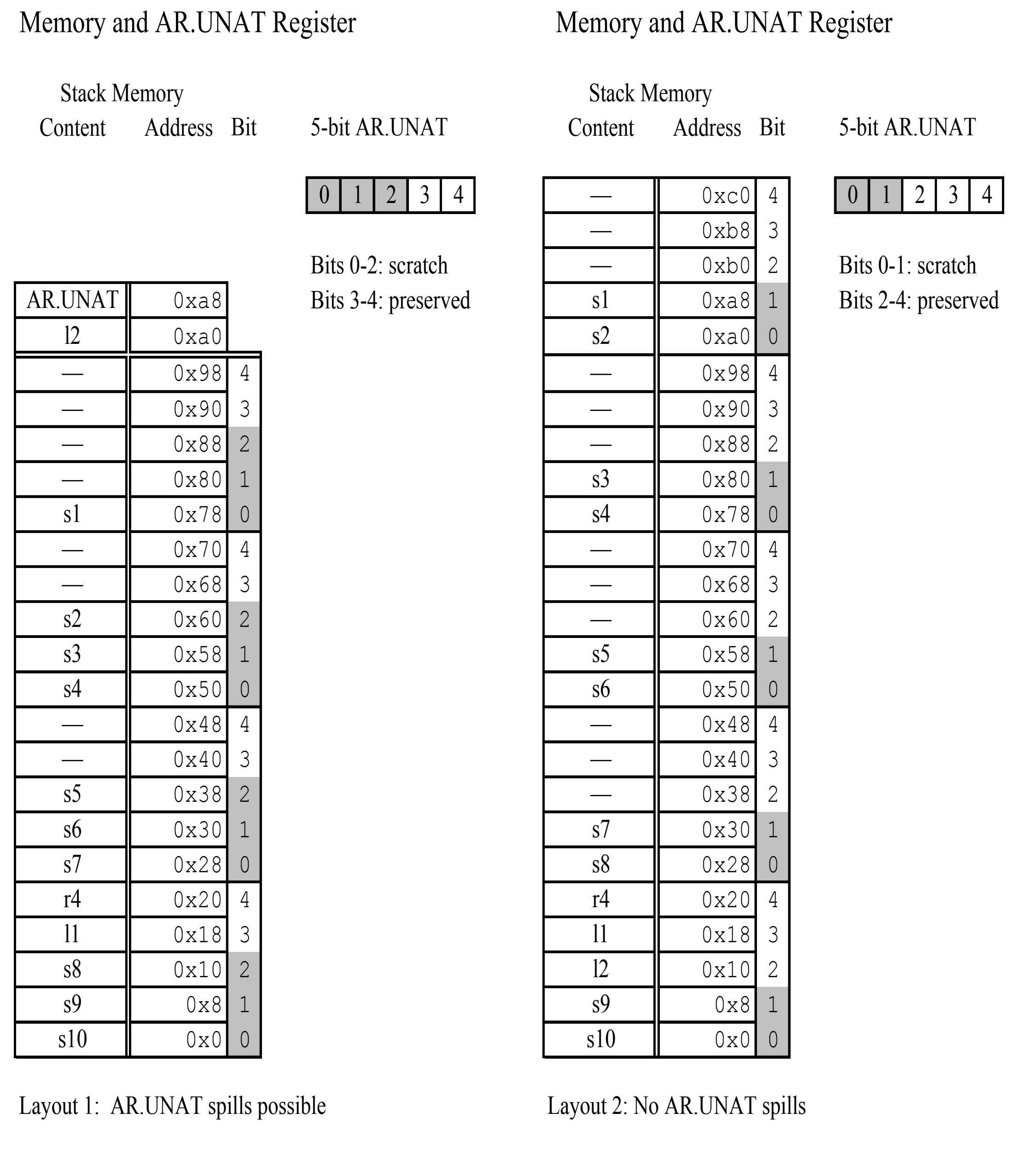
NaT bit of the spilled (filled) live range is in the spilled (filled) AR.UNAT (Figure 42). In this naïve approach the entire AR.UNAT register is spilled to save and restore one NaT bit. Spilling of the AR.UNAT register could be avoided, if the compiler ensured that no more than 64 live ranges with a potentially set NaT bit interfere, but control speculation in the compiler is register pressure unaware.

While general NaT propagation enables more efficient recovery code, it can result in many spills of the AR.UNAT register, since the NaT bit must be potentially preserved for any symbolic register in the dependence chain originating at an ld.s. If such a symbolic register is spilled, its NaT must be available at a fill. If the compiler requires that the ld.s destination and the chk.s *must* match, modeling NaT propagation becomes simpler. Then only the NaT bit of a speculated load destination must be preserved. When any other speculated live range (in the dependence chain of an ld.s destination) is spilled, a st8.spill must still be used to avoid a NaT consumption fault at any program point where its NaT may be set. But the NaT bit does not need to be preserved and a regular load—rather than an ld8.fill—can be used to load the value. This means that for most speculated live ranges the NaT bit value can be ignored and does not need to be preserved (in the case of spilling) in the AR.UNAT application register. The next section describes the algorithm that exploits "matching" ld.s and chk.s.

7.1.2 An advanced NaT Propagation Algorithm

Modeling general NaT propagation in the compiler is challenging. It is conceivable to develop a predicate-aware NaT propagation algorithm based on available NaT and live NaT data flow algorithms, similar to available variable and live variable algorithms used for live range approximation. But two observations give a simpler method for NaT modeling. First, under the assumption that each ld.s destination has a matching chk.s source (in other words, they are the same virtual or symbolic register), the compiler does not need to model the NaT propagation of the processor entirely. Instead, it can model NaT propagation only for the destination registers of ld.s. Second, the compiler can partition the bits of the AR.UNAT application register into two classes: preserved and scratch. A preserved bit corresponds either to the NaT bit of a static preserved general register (r4, r5, r6 or r7 in Figure 4, p.22) or to the NaT bit of a destination register in a speculated load. The scratch bits correspond to any other register that may have its NaT bit set, like a symbolic registers in the dependence chain of a speculated load. Ld.s destinations are the only NaT producers. So only for them (and the used preserved registers r4-r7, which are spilled at function entry and filled at function exit(s)) the NaT bit must be preserved in case of a spill and restored at a fill. To filter the symbolic registers that need only a scratch NaT bits it is sufficient to mark instructions in the dependence chain from the ld.s to the matching chk.s. This can be done either in the scheduler or the chain can be recomputed in a separate phase before register allocation. When a symbolic register in a marked instruction is spilled, a st8.spill must be used. In case of a fill, only when the symbolic register is the destination of a NaT producer (ld.s) and its NaT bit could be checked by a following chk.s, an ld8.fill must be used. To find all places where an ld8.fill must be used we introduce the concept of a live NaT. This is a sub-live range of a ld.s destination where the NaT could be set. Live NaTs start at ld.s and end at chk.s. A live NaT analysis similar to a live variable analysis gives all program points of the live NaT range. We note that the compiler can use a pseudo ld.s and chk.s instructions to model general NaT propagation under the restriction that ld.s must have a matching chk.s. The pseudo chk.s instruction marks the end of one NaT range and the pseudo ld.s instruction the beginning of another. Although the actual ld.s and chk.s mismatch like V7 and V9 in Example **7**, the pseudo instructions partition the speculated live ranges so that the live NaT analysis is applicable.

Figure 43 illustrates the effect of the algorithm and choices for stack memory layout for a hypothetical 5bit AR.UNAT register. For illustration we assume one preserved register r4, which is spilled and filled at function entry and exit respectively, 10 “scratch” spills s1, …, s10 for candidates that may set a NaT and two interfering ld.s destinations l1 and l2. Assume s1,…, s10 interfere with l1 and l2. In layout 1 r4 and l1 are associated with AR.UNAT bit 4 and 3 respectively. Bit 4 cannot be overwritten, since it must be preserved for the entire function. Thus no other st8.spill can go to a memory address that would write the NaT of its source register to this bit. For bit 3 the situation is different: a non-interfering candidate can be written to bit 3 and even stored at the same memory address, but an interfering live range must ensure not to destroy bit 3. Therefore the memory address corresponding to bit 3 can be used when the allocator determines that there is no conflict with l1.



Memory and AR.UNAT Register

Stack Memory

Content	Address	Bit
AR.UNAT	0xa8	
l2	0xa0	
—	0x98	4
—	0x90	3
—	0x88	2
—	0x80	1
s1	0x78	0
—	0x70	4
—	0x68	3
s2	0x60	2
s3	0x58	1
s4	0x50	0
—	0x48	4
—	0x40	3
s5	0x38	2
s6	0x30	1
s7	0x28	0
r4	0x20	4
l1	0x18	3
s8	0x10	2
s9	0x8	1
s10	0x0	0

5-bit AR.UNAT

0	1	2	3	4

Bits 0-2: scratch
Bits 3-4: preserved

Layout 1: AR.UNAT spills possible

Memory and AR.UNAT Register

Stack Memory

Content	Address	Bit
—	0xc0	4
—	0xb8	3
—	0xb0	2
s1	0xa8	1
s2	0xa0	0
—	0x98	4
—	0x90	3
—	0x88	2
s3	0x80	1
s4	0x78	0
—	0x70	4
—	0x68	3
—	0x60	2
s5	0x58	1
s6	0x50	0
—	0x48	4
—	0x40	3
—	0x38	2
s7	0x30	1
s8	0x28	0
r4	0x20	4
l1	0x18	3
l2	0x10	2
s9	0x8	1
s10	0x0	0

5-bit AR.UNAT

0	1	2	3	4

Bits 0-1: scratch
Bits 2-4: preserved

Layout 2: No AR.UNAT spills

Figure 43 5-bit AR.UNAT Register and Stack Memory Layout Options

The scratch spills interfere, but they can share the same NaT bit (bit 0 – bit 2), since for them the NaT bit does not need to be preserved. It only needs to be saved to avoid a

NaT consumption fault. To save the NaT bit of l2, the entire AR.UNAT register is spilled as in the conservative method. But, unlike for the conservative method, for s1 to s7 the AR.UNAT register is not stored, which saves 7 spill/fills and 14 AR.UNAT moves. In the second layout one more AR.UNAT bit is preserved. This saves the AR.UNAT spill (and 2 AR.UNAT moves) for l2 at the expense of larger stack memory.

In layout 2 three bits (3, 4 and 5) in the AR.UNAT are used as preserved bits. In this scenario l1, l2 and r4 can preserve their NaTs. Since there are only two scratch bits, which s1,…,s10 can use, the memory stack increases compared to layout1, but spills of the AR.UNAT can be avoided.

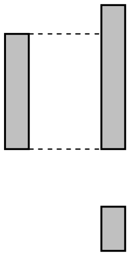
The algorithm applies also to a live range that combines control-and data speculation. Data speculated live ranges have their own unique characteristics that can be exploited with special register allocation techniques.

7.2 Data Speculation

IA-64 provides an advanced load (ld.a) and two advanced load check (chk.a, ld.c) instructions for data speculation. This enables the code generator to schedule a load across a potentially overlapping store (*breaking the store barrier*). At execution, an advanced load records information about its physical destination register, memory address and data size in the Advanced Load Address Table (ALAT) [13]. If a subsequent store overlaps, then the hardware invalidates the corresponding ALAT entry to indicate the collision. As with control speculation, the compiler generates recovery code that will execute only in case speculation fails (Example 8). There is something peculiar about data speculated live range, which the register allocator can exploit to reduce its interferences. Data speculated live ranges that end in a chk.a have a range where interferences with non-alat live ranges can be *ignored*. Since the chk.a only checks the register number, but not the value in the alat, the chk.a destination register can be shared with a non-data speculated candidate in a special circumstance. If the non-speculated candidate has a live range that is contained entirely in the range from the chk.a to the penultimate use of the data speculated live range, the interferences of the two live ranges can be ignored and both candidates be assigned the same register. We call the section of a data speculated live range from the ending chk.a to its penultimate use the *ALAT shadow*.

Rather than building a containment graph (Cooper and Simpson [24]), the Itanium compiler associates an *shadow* ALAT live range to each data speculated live range ending in a chk.a. It uses pseudo instructions for modeling the shadow, i.e. the live range section from the chk.a (which is the last use and ends the live range) to its penultimate use (if it exits). In the shadow all interferences with regular live ranges (not data speculated) get ignored. Interference with another data speculated candidate must be taken into account, since two overlapping data speculated live ranges cannot share the same register. Example 8 gives an illustration. Live range V4 is data speculated. The load in line 5 is hoisted above the store in line 4. The dependent add in line 6 is speculated also. We assume there is no use of V4 after the chk.a, so the live range for candidate V4 ends at the chk.a. A shadow live range V4' modeling the ALAT shadow for V4 is introduced ranging from the chk.a to the penultimate use at the "add" in line 2.

```
      Original Code        Data Speculated Code      Pseudo Code   LR of V4

 1:                              ld8.a  V4=[V1]
 2:                              add    V5=V4,V6     puse V4'
 3:                              ...
 4:   st4   [V10]=V11            st4    [V10]=V11
 5:   ld8   V4=[V1]              chk.a  V4, rec      pchk V4'
 6:   add   V5=V4,V6       cont: ...
 7:                              ...
 8:                        rec:  ld8    V4=[V1]
 9:                              add    V5=V4,V6
10:                              br     cont
```



Example 8 Data Speculation with Recovery Code and Alat Live Range

The modeling of the ALAT shadow is accomplishing by the use of pseudo instructions. Again, Example 8 shows the pseudo instructions "puse" and "pchk". The live range for candidate V4 has two components: First, it starts at the advanced load in line 1 and ends at the chk.a in line 5. Second, the non-speculated dependence chain in the recovery code from line 8 to line 9. The shadow live range V4' ranges from line 2 ("puse") to line 5 ("pchk").

8 Scalable Register Allocation

A coloring allocator is usually a fast and efficient compiler phase. It can cause a compile time problem because of memory consumption due to a large number of candidates. To address the memory problem multiple strategies can be employed. This section discusses the pros and cons of these methods, and proposes scalable register allocation, which can solve the allocation problem in general for an arbitrary set of register candidates. Using a scalable allocation scheme, coloring allocators can handle fast and efficiently any large-size allocation problem.

The register allocator in the Intel Itanium compiler supports a variety of allocation strategies:

1. Region based allocation
    a. This is the default strategy in the compiler. A region is either a loop or outer acyclic region.
2. Conversion of local candidates to global candidates
    a. The register candidates consist of local and global candidates. For local candidates all references are within a basic block. All locals are numbered consecutively. For example, when the first basic block has local virtual registers v1-v10, the first local in the next block will be v11. When the allocator recognizes that the number of locals exceeds an internal threshold for the entire control flow graph, it will hash locals to global virtual registers effectively reducing the overall number of register candidates. Hashing is interference agnostic, but locals in one basic block cannot hash to the same (new) global. Only locals from different basic block can hash to the same global. The method works since the number of locals per basic block is limited to a few hundred candidates. Therefore the thousands of locals in the entire routine can be hashed to relatively few new global candidates at the cost of extra interference: the interferences of a hashed global variable are the union of all interferences of the locals that have been hashed to it.

3. Basic block allocation only
    a. In this approach the candidates are allocated per basic block. Global virtual registers are always spilled at basic block boundaries. No dataflow analysis is employed. This approach can be seen a simple variant of a region based allocator, where the regions are simply basic blocks.

All methods can reduce memory usage and compile time, but they have disadvantages as well. Region-based methods rely on global dataflow analysis, which itself can cause compile time problems. These methods also have the cost of maintaining global information necessary to reconcile allocations in different regions at the region boundaries. In case of loop based region allocators, they may not be able to handle large routines that have no loops. In the case of basic block allocators that spill global candidates at block boundaries they can cause huge performance regressions. Hashing local candidates and replacing them by global virtual registers reduces the number of register candidates and therefore the size of the interference graph, but is useful only when local candidates by far outnumber global candidates.

Functions generated in the framework of large server applications may contain hundreds of thousands of global register candidates in a single, loop-free procedure and expose the memory problem. Among the methods above, only the third method can be used to compile the application in reasonable compile time, but at a potentially big performance cost. This thesis proposes scalable coloring allocation that can be applied to procedures with any number of global variables.

The scalable register allocator is based on the observations that a coloring allocator solves small to medium sized allocation problems in almost linear time and space. Scalability is then achieved by a two step process. First, the scalable allocation partitions the set of register candidates directly. Second, it runs the coloring allocator on each set. Effectively it partitions the original allocation space into many small subspaces that can be solved quickly. Rather than solving the original allocation problem for all candidates at once, the algorithm avoids exponential compile time increase by partitioning the candidates into smaller subsets. Using the ideas in this chapter we show that a coloring allocator can be parallelized and allocate independent (disjoint) sets of variables in parallel.

```
Input: $S = \{S_1,....S_n\}$ , a set of sets of candidates

procedure scalable_serial_allocation()
      foreach $S_i$ in $S$
            Allocate($S_i$);
      endfe
      AllocSpillMemory();

endproc

procedure scalable_parallel_allocation()
      - Partition physical or virtual target registers:
      $R = \{R_1,...,R_j\}$ .
      - Partition candidate set into sets of j subsets, where j
      is the number of register partitions:
      $S = \{S_1,...,S_j, S_{j+1},...,S_{2j},...,S_{(n-1)j}...,S_{nj}\}$

      while  $S \neq \emptyset$  do
            select worklist $W = \{S_{x_1},..,S_{x_j}\}$ ;
            PAllocate$(W, R)$ ;

            $S = S \setminus W$ ;
      od;
      AllocSpillMemory();
endproc
```

Figure 44 Serial and Parallel Scalable Register Allocation

Figure 44 shows the high level view of the scalable allocator. It may use techniques like live range splitting and pre-materialization in a preparation phase. This phase may also renumber the virtual register candidates in case of a hierarchical allocator that uses virtual rather than physical colors. The hierarchical allocator will be discussed in more detail later in this section. After renumbering a selection phase uses filters to partition the candidates are partitioned. There are many possible choices for a filter. Some specific examples for a filter are:

- Select all candidates of same type, e.g. all floating point or all integer candidates. This specific filter has been implemented in the Intel compiler. Floating-point candidates must be either allocated first or together with integer candidates, since spilling of a floating-point candidate introduces new integer candidates.

- Select candidates referenced in certain regions, unless they have been selected already. For example, all candidates in innermost loops can be picked. The difference to a region-based allocator is that the scalable allocator would assign the same register to a candidate across the entire routine. This is not necessarily so for a region-based allocator. It may assign different registers to the same candidate in different regions. The assignment is then reconciled (by adding reconciliation code, i.e. mov, store or load) at region boundaries.
- Select candidates based on profile data. When basic block profile data is available, one choice is to pick "hot" candidates first, where hotness is determined by a heuristic threshold based of reference frequencies of the candidates. All hot candidates are allocated first, and if a candidate is assigned a register, the assignment -unlike in a region-based allocator- holds in the entire routine. Selection by hotness is an idea that goes back to the Chow allocator (at least).

The outcome of the selection phase is a set S of n partitions $S_1,...,S_n$. Each partition is a set of candidates. The number of candidates per partition can vary. They don't have to be the same in each partition. The key is to control the number of candidates in each partition so the coloring allocator can solve the allocation problem for the partition efficiently. In the result section we will analyze a simple partition scheme where each partition contains N/n candidates, where N is the total number of candidates and n the number of partitions.

In the serial version the register allocator is invoked for each set of candidates. The full register set available (except for reserved registers and spill registers) is used for allocation. After allocating the first partition, all candidates are mapped to physical registers. Subsequent allocations will not change physical registers and take interferences with physical registers into account. Internally access functions in the allocator and dataflow routines check that a specific candidate is in the selected candidate set. Candidates that are not in the set get ignored. Candidates in the set interfere only with each other or physical registers. Each invocation of the register allocator uses symbolic stack memory accesses for spill code. Therefore the memory stack must be finalized after the last set of candidates has been allocated, because only after the last allocation all stack references can be resolved. In Figure 45 is an example for a serial scalable allocator assuming eight physical register r1, …, r8. In the example, two sets have been selected and the partitions are:

$$S = \{S_1, S_2\} = \{\{V1, V2, V4999\}, \{V5000,..., V9999\}\}$$

The first instance of the allocator assigns e.g. r1 and r2 to V1 and V2 respectively, r8 to V9999 and spills V1000. When the allocator is invoked for the second partition, V9999 e.g. is assigned r8, since V9999 does not interfere with r8. After the second and final partition, spill addresses can be allocated in the memory stack.

Candidates	1st Allocation		2nd Allocation		Result
V1	V1	r1	r1	r1	r1
V2	V2	r2	r2	r2	r2
…	…	…	…	…	…
V1000	V1000	spill	spill	spill	spill 16
…	…				
V4999	V4999	r8	r8	r8	r8
V5000	V5000	ignore	V5000	spill	spill 24
…	…		…	…	…
V9999	V9999	ignore	V9999	r8	r8

Figure 45 Example for Serial Scalable Allocator for registers {r1, …,r8}

The parallel version of the scalable allocator is configurable depending on the number of candidates and processors/cores available. Also, parallelization can be achieved in many ways. For example, the physical register set can be partitioned $R = \{R_1,..., R_n\}$ into n sets. Similar the set of candidates can be partitioned into n or a multiple of n sets: $S = \{S_1,..., S_n, S_{n+1},.., S_{kn}\}$. In this case registers in R_i get assigned to candidates in $S_i, S_{i+n},..., S_{i+(k-1)n}$. Partitioning ensures that the candidates and the physical registers are compatible. This means, for example, that a set S_i of floating point candidates is allocated to a partition of physical floating point registers R_i. In Figure 46 a simple register file with general registers r1,..,r4 and floating-point register f1, f2 is partitioned into three sets. Any filter as described above may be used to partition the candidates.

Global candidates V1-V10 can be assigned r1 or r2, V11-V100 can be assigned r3 or r4, and V101-V120 and local candidates, v1- v9, can be assigned f1 or f2. The PAllocate() routine will then span three allocations $(S_1, R_1), (S_2, R_2)$ and (S_3, R_3) in parallel.

$$R = \{R_1, R_2, R_3\} = \{\{r1, r2\}, \{r3, r4\}, \{f1, f2\}\}$$
$$S = \{S_1, S_2, S_3\} = \{\{V1,...,V10\}, \{V11,...,V100\}, \{V101,...,V120, v1,.., v9\}\}$$

Figure 46 Example for Partition of Register File and Candidates

Partitioning the physical register file could be too restrictive. Since relatively few physical registers are available, either the number of partition sets or (and) the number of registers available per partition set must be restricted. The number of partition sets is an upper limit to the parallelism available for an allocation problem. Partitioning could also result in avoidable spill code. Since the registers in the partition sets must be disjoint, a single partition has relatively few registers available for assignment. But these restrictions can be worked around by staging the allocation. A more general method is using virtual registers for each partition in a first allocation step. This reduces the size of the allocation problem, unless the interference graph is a complete graph. In the second and final step, the allocation algorithm assigns physical registers to all candidates. The point of this two step (or hierarchical) allocation scheme is that in the first step no physical register is committed and the number of candidates is reduced by virtual allocation. The original interference graph does not have to be built. In this scheme no limit of virtual target registers is assumed. A suitable set of virtual colors may be chosen for each candidate set and all virtual target registers can be assumed to be disjoint enabling parallel incarnations of the allocator. Figure 48 illustrates the concept. The candidates are split into three integer sets and one floating-point set (Figure 47). They are allocated in parallel to virtual target registers. In the example, there are 768 integer and 128 floating point virtual target registers. Therefore, after the parallel allocation, the allocation problem has been reduced to 896 candidates. The target size of the virtual register partition can be dynamically chosen so that spill code in the parallel step can be avoided. This is evident from the fundamental inequality (Equation 4, p. 54): for a candidate partition the maximal number

M of neighbors in the interference graph can be determined for the set before coloring. Forming a set of M+1 virtual target register is then sufficient to color the set.

$$R = \{R_1, R_2, R_3\} = \{\{ivr1,...,ivr256\},\{fvr1,...,fvr256\},\{ivr257,...,ivr512\}\}$$
$$S = \{S_1, S_2, S_3\} = \{\{V1,...,V75000\},\{V75001,...,V150000\},\{V150001,....V225000\}\}$$

Figure 47 Example for Virtual Register and Candidate Partitions

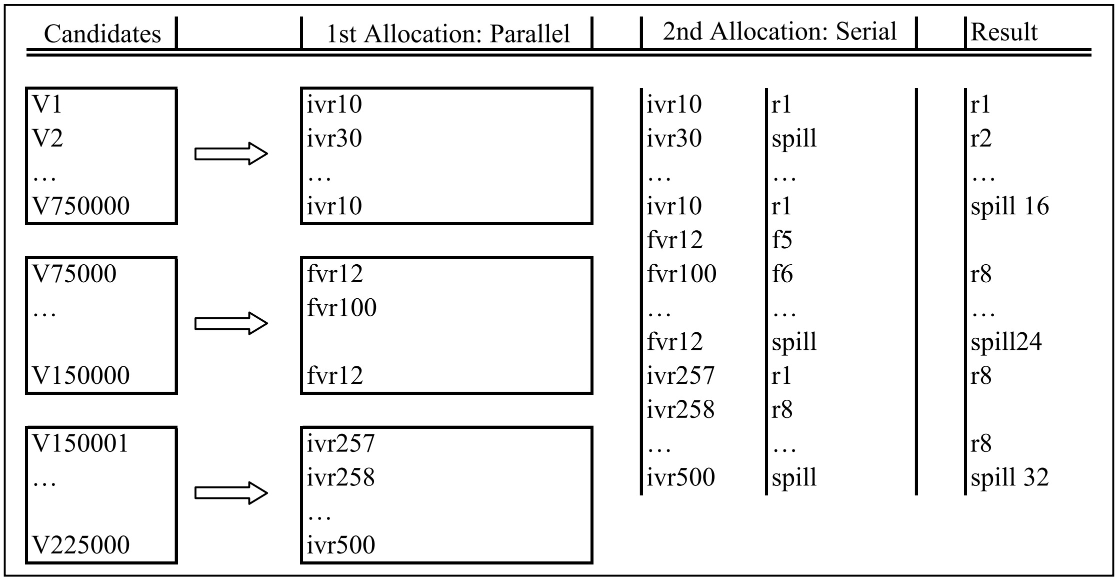


Figure 48 3-Way Parallel Scalable Allocator with Staged Allocation

There can be more partitions in the set S of candidates than in the set R of registers, but the number of partitions in R is the maximal number of parallel allocations. After all candidate sets have been allocated in parallel, the virtual colors get allocated to physical registers in a final round of allocation, but on a much smaller problem space. Like in the serial configuration the parallel scalable allocator must take care of the memory stack layout in a post-allocation phase. It is conceivable to allow multiple stages of allocation with virtual target registers.

In summary the scalable allocator partitions the set of candidates. Partitioning can as simple as dividing the candidates (of one class, e.g. integer or floating-point) into equal subsets. Since the candidates are partitioned before allocation, data flow analysis runs only for the selected subset (and physical registers). In the serial allocation scheme the allocator is invoked one subset of a time. The input to invocation N+1 is the output of invocation N. Invocation takes into account interferences with physical registers that have been assigned in previous invocations. In the parallel allocation scheme the physical registers are partitioned also. Allocation can be run for subsets of candidates. There can be as many allocations run in parallel as there are partitions of the physical registers. In the parallel scheme interferences between subsets of candidates can be ignored since they are guaranteed to get assigned different registers. In either scheme (serial or parallel) a hierarchical allocator can use virtual registers to avoid spills at the first stage of allocation and reduce the number of candidates.

9 Related Work

This thesis describes extensions of a coloring allocator covering features provided by the Itanium architecture like dynamic register stack, control- and data speculation and predicated code. The idea of dynamically resizing the register stack is in the original IA-64 design, and can be seen in the resize semantics of the alloc instruction (Intel manuals [42]). The first description of compiler algorithms exploiting the semantics of the alloc instruction and applying it to general regions is Hoflehner and Pierce [38]. Douillet et al. [27] evaluated the idea in the ORC compiler on paths with high register pressure, but they measured negative results on six CINT2000 benchmarks: although they saved RSE traffic, they measured a slowdown of 27.34% in the geomean and suggested the cost of the alloc instruction is responsible for it. Other papers come to different conclusions. Settle et al. [69] apply multiple alloc to call sites and show static improves, although the run-time of CPU2000 benchmarks did not improve for their prototype implementation in the Intel Itanium compiler. For the production version of the algorithm Hoflehner et al. [39] report a 1% speed-up on Oracle TPC-C. This thesis gives a review of this algorithm and shows results for an improved version on the CPU2006 benchmark suite. The gains for this suite are still at about 1%, but this has to be seen in perspective. Overall the performance cost of RSE traffic is much less than 10% (Desai et al. [26]), so a 1% performance run-time performance gain is from reducing the dynamic RSE traffic by more than 10%. Yang et al. [78] take an interprocedural approach, which they call the “quota assignment problem”. The observation is that when RSE traffic kicks in for the function on the call stack, it might be cheaper by actually spilling some register in the compiler and reducing the stack frame size. In this form it is somewhat reminiscent of Chow’s shrink-wrapping approach [21]. Yang et al. report improvements from their interprocedural approach on two CINT2000 benchmarks, perlbmk (13%) and crafty (3.7%). Their numbers are for the ORC compiler, for which they report very high RSE traffic number compared to the Intel Itanium production compiler [26]). They don’t compare their method with a multiple alloc implementation for call sites, which might have given them performance gains at a smaller implementation cost. Weldon et al. [75]

is studies the effectiveness of the RSE and proposes hardware mechanisms to reduce RSE spills and fill.

Register allocation for speculated code seems to be neglected in literature. The focus there is on headroom studies for performance gains from speculation (e.g. Wu et al. [77]) and general frameworks to handle recovery code (e.g. Lin et al. [53]). One major purpose of these papers is to expose speculation at a higher level IR, pass it on to lower level IR and show how classical optimizations like PRE can exploit it. On the other hand, this thesis shows how to extend a coloring allocator to handle control- and data speculated code effectively. Our concern about control speculation is program correctness. In particular the compiler needs to generate code that cannot cause a NaT consumption fault at run-time. With respect to data speculation, this thesis describes how the allocator can reduce interferences for data speculated live ranges that end in chk.a and have a penultimate use. [4] This idea was also mentioned in Bharadwaj et al. [9], but not described in detail.

There is a rich body of literature covering predication and representing predicate relations. The IMPACT compiler uses the Predicate Hierarchy Graph (PHG) (Mahlke et al. [56]). For each definition of a predicate the PHG tracks the predicates that guard the definition. It can also handle basic Boolean expressions and is applied to analyze predicated code of a hyperblock. A hyperblock is a predicated superblock, which is an acyclic single entry multiple exit region in a control flow graph. The PHG is mainly used to derive predicate disjointness. Rather than applying the PHG for predicate-aware dataflow analysis, the IMPACT compiler uses reverse if-conversion (Warter et al. [74]) to convert predicated code to control flow. On this graph it performs classical data flow analysis. A more sophisticated approach than the PHG is the predicate query system (PQS), which is based on the predicate partition graph (PPG). For PQS the key references are Johnson and Schlansker [45] and Gillies et al. [31]. Johnson and Schlansker [45] focus on the analysis of predicated code. They model predicate relations in terms of relations between execution sets and introduce the predicate partition graph to answer queries about predicate relations. PQS can determine accurately predicate relations that

[4] If a live range ends is chk.a but has no penultimate use, an ld.c (load check) should have been used instead. But IA-64 architecture does not require the use an ld.c.

can be expressed as logical partitions. Two predicates P2 and P3 form a predicate partition P1 when P1 is the union of P2 and P3, and P2 and P3 cannot both be true simultaneously (=at the same program point). In their paper Johnson and Schlansker [45] show how to build the PPG starting from predicated code, give pseudo-code for the PQS queries-including lub_sum and lub_diff (see Appendix 12.3, p.141) and apply their system for predicate-aware live variable analysis. But their paper has no experimental evaluation of their method. Gillies et al. [31] can be seen as the follow up to Johnson and Schlansker [40]. Their paper shows how to construct the predicate partition graph from the control-flow graph and discusses in some detail predicate-aware live variable analysis and predicate-aware interference construction. They show that predicate-aware allocation reduces register pressure for predicated code *statically* between 20% -70% on 23 procedures picked from the SPEC92 benchmark suite. Their paper has no run-time data. In preparation of PPG construction they add basic blocks (JS-blocks) on critical edges, but they only claim to so to "*simplify* the creation of partitions" (p.118). We showed that the blocks increase the accuracy of predicated analysis (cf. discussion of partition and overlap live ranges in Chapter 6.3, p.83). Gillies et al. [31] also discuss the start of a live range problem (see Figure 15) and mention a case where predicate-aware data flow must be conservative across back edges, which is similar to our example in Figure 28. Since PQS bases analysis can determine the start of a live range, PQS papers do not discusses predicate-aware available variable analysis. In the classical case (= no predication) the purpose of available variable analysis is determining the start of a live range. For acyclic, if-converted regions predicate partitions are sufficient to derive this information. Our approach is close to Gillies in the sense that we assume predicated code is derived from compiler generated control flow graphs. But rather than focusing on the predicate analysis system, we ask the basic question of what kind of live ranges exist and what information is necessary to model their interferences in the presence of predicated code. We identified four types on predicated live ranges – match, dominate, partition and overlap –and two methods for interference modeling– "simple" and "complex", and three implementation strategies. The simple method models interferences of all "match", "dominate" and some "partition" live ranges precisely, and remaining live ranges conservatively (Theorem 6-1). The complex method models all live range precisely

(Theorem 6-2). In implementation strategy 1, only match and dominate live ranges are modeled precisely. We show that this method gives run-time performance compared to PQS at lower compile-time and-implicitly-lower implementation and maintenance cost. Strategy 2 models all simple live ranges, which include all “match”, “dominate” and some “partition” live ranges, precisely, and strategy 3 models interferences of all live range precisely. Strategy 3 is equivalent to a PQS based implementation when predicated code is derived form control-flow code. We don’t build extra structure like a PPG and use “classical” techniques only to handle different classes of predicated live ranges. But PQS based allocator, which includes the PPG, is used as a reference implementation to show the effectiveness of our method. Our result depends on two parameters: First, the if-converter is conservative and if-converted regions are relatively simple. This may give a bias to “match” and “dominate” live ranges. Second, an increase in register due to handling “partition” and “overlaps” live ranges conservatively (by allowing false interferences for them) can be tolerated by the (relatively to other architecture) large register file of IA-64. When the two assumptions are not valid, the choices are to implement strategy 2 or strategy 3 or a scheme like PQS. The implementer has also the option to pick and choose different allocation strategies for different function or even region at compile-time. The core of Chapter 6 that discusses register allocation for predicated code can be found also in Hoflehner [40].

More subtle predicate analysis methods that derive accurate predicate relations for already predicated code have been developed also. Eichenberger [28] represents logical predicate relations, so called P-facts, and determines predicate relations, in particular predicate interference, using a logic solver. He applies this information for register allocation in a hyperblock. Sias et al. [71] developed the predicate analysis system (PAS), which is as accurate as Eichenberger’s, but can determine predicate relations globally using a BDD solver. Eichenberger’s and Sias approach is evaluated in a research environment. It is not clear that their methodologies are practical for a production compiler. In contrast this thesis makes no attempt to address the general predicate relation problem, based on the premise that the key predicate relation the coloring allocator needs is predicate disjointness. Two predicates are disjoint if they are not true at the same program point. Since predicates materialize as block predicates, this information can be derived directly from the control flow graph. Our allocator is built on the assumption that predicated code is derived from acyclic control flow graph regions. If this assumption were violated, the allocator would still produce correct results by assuming all "unknown" predicates interfere with each other an the block predicates.

Coloring allocators cannot handle allocation problems of any size. Too resolve this compile time problem many methods have been devised to reduce the allocation space. The ideas revolve around partitioning a given routine into regions (e.g. Hank [35]) or partitioning the interference graph. Callahan and Koblenz [15] describe a general region allocation scheme ("Hierarchical Graph Coloring"). They partition the control flow graph into a set of tiles. Tiles are sets of basic blocks with additional properties, so that a tile tree can be constructed: two tiles are either disjoint or contained (tile 1 is subset of tile 2 or vice versa) and there is a single root tile. Then graph-coloring is applied to each tile (region) in a bottom-up walk of the tree. At tile boundaries the allocations are reconciled. Reconciliation is necessary since a live range L1 may get assigned register r1 in tile 1 and register r2 in tile 2. At the tile boundary of tile 1 and tile 2 a "reconciliation" move from e.g. r1 to r2 must be inserted. Hierarchical graph coloring covers loop trees and graph-partitions based on single-entry single-exit ("SESE") regions. It is noteworthy that their allocator uses "pseudo" registers, which are assigned physical registers in a reconciliation phase. Norris and Pollack [61] pursue region based allocation in a similar fashion, but based on the program dependence graph ("PDG"). Statements guarded by the same control statement form an allocation region. Their region may have multiple exits. Fusion-based allocation partitions the control flow graph into arbitrarily disjoint regions (Lueh et al. [55]). The idea of the fusion allocator is to delay spilling until the interference graphs of two simplifiable regions get fused. When the combined graph would be no longer simplifiable, the fusion operator, based on feedback profiling information, attempts to split live ranges in order to minimize spill code at the region boundaries. In all these methods the allocation problem is partitioned into a set of sub-problems that can be solved more efficiently than the original problem. The cost of the methods is extra book-keeping to glue together the local (per region) solutions at boundaries. But there are functions generated in the framework of large server applications that exhibit hundreds of thousands of register candidates. Region based methods are usually focused on loops, but can be tuned to handle general regions and partition large programs. But they also rely on global dataflow computation, which itself can consume a significant amount of memory. In addition, they rely on global data structures to reconcile allocations at region boundaries. Also, it seems a hard problem to parallelize region-based allocators without synchronization in form of region reconciliations.

Gupta et al. [33] proposed clique separators for partitioning the interference graph. A clique separator is a clique that partitions a graph into two disjoint components. The clique allocator computes spans (definition-use chains) and identifies a set of clique separators. Each span can be contained in (at most) a fixed number of sub-graphs. Each sub-graph is colored separately, while it includes the nodes of a separating clique. The final graph is composed from the sub-graphs possibly with renumbering of assigned registers and spilling (or register copies) at separator boundaries. Given n nodes and m clique separator the clique separator consumes $O(n^2 / m^2)$ space and $O(n^2 / m)$ time. Like for region based methods, additional cost is from extra book-keeping to glue together the allocations for the sub-graphs and rely on global dataflow computation, which itself can consume a significant amount of memory.

In contrast this thesis proposed a family of scalable allocators, which do not rely on region shapes and take the scalability of dataflow algorithms into account. Since the focus is on partitioning the symbolic registers before allocation, the dataflow analysis to determine live ranges needs to consider only the candidates contained in that partition and physical registers. A scalable allocator can be engineered to solve the allocation problem in parallel and does not need boundary reconciliation. It is capable of allocating effectively programs with very large sets of register candidates. We showed how a scalable allocator can be used also for parallel allocation e.g. on multi-core machines, so each core can run an “allocation thread” for a subset of candidates. Parallel allocation applies in particular to machines with large register files, since the register files can be partitioned and each allocation is associated with one partition. This guarantees that candidates in different partitions are independent: since they get assigned different registers, interferences between classes of candidates can be ignored. Scalable allocation can result in higher spill cost and performance degradation (assuming the allocation problem can be solved in a classic coloring allocator) since the register file is partitioned a priori and in fact more registers may get used than necessary. This can hurt e.g. IA-64 in from with higher RSE traffic. Scalable allocators can be seen as extensions of candidate splitting ideas. Splitting the candidates is often implicit in the coloring heuristics. Well-known examples include coloring basic block local candidates first, then the global candidates (e.g. Briggs [12]) or dividing the candidates into classes, like floating-point and integer candidates (see e.g. Chapter 4). We demonstrated the potential of a simple allocator that divides the candidates into constant subsets and allocates one subset at the time. It can solve effectively allocation problems for the "f_serverapp" test case, which has more than 500K candidates in a single function (Section 10.2.3).

10 Results

We obtained the performance data on a 1.6 GHz Montecito processor using the Intel Fortran/C++ optimizing product compiler (version 11.1, 2009). The detailed configuration is listed in Table 8. The benchmark suite is CPU2006, a popular industry-standardized CPU-intense suite used by OEMs for stressing a system's processor, memory subsystem and compiler.

Table 8 Experimental Setup

Processor	Intel Itanium 2 (Montecito) Processor, 1.6 GHz
Compiler	Intel Fortran/C++ Compiler (Version 11.1)
Memory	4 GB Main, 16 K L1D, 16KB L1I , 256K L2D, 1M L2I , 12M L3 D+I
OS	Red Hat Enterprise Linux AS Release 4 (Kernel 2.6.9-36.EL #1 SMP)

CPU2006 has two sub-suites. The integer suite (CINT2006) consists of twelve benchmarks containing general user applications like compiler, interpreters, games, simulators and database. A rough overview of the CINT006 benchmarks can be found in Table 9. The floating-point suite (CFP2006) contains a set of 17 applications relevant in high-performance computing including equation solvers, physical, chemical and biological system simulations, speech recognition systems and a ray tracer. More details about the CFP2006 benchmarks are in Table 10. To give a better idea about the relative complexity of the benchmark, tables Table 9 and Table 10 show the lines of code (LOC), compile times normalized to the compile time of 470.lbm and the number of routines (HR = hot routines) where over 90% or more of the total benchmark execution time is spent. As an example, the normalized compile time of 470.lbm is 1. It takes 189 times as long to compile 403.gcc. The number of hot routines varies a lot for the benchmarks. For example, 456.hmmer has only a single hot routine where 90% or more of total execution time is spent. This is a characteristic that suggests 456.hmmer behaves more like a HPC (high-performance computing) application which tend to have a few hot loops. On the other hand, CFP2006 which represents floating-point computing intense (and thus HPC) applications has benchmarks like 481.wrf, where 90% or more of the total execution time is spread across 33 routines.

Table 9 SPEC CPU2006 Integer Benchmarks

Benchmark	LOC	CT	HR	LANG	Description
400.perlbench	155432	47.2	33	C	Based on perl V5.8.7
401.bzip2	8293	4.2	8	C	Based on bzip2 V1.0.3
403.gcc	518781	189	61	C	Based on gcc V3.2 generating Code for Opteron
429.mcf	2685	1.6	2	C	Single-Depot Scheduling in Public Mass Transportation
445.gobmk	197215	40.2	39	C	Go Game Execution and Position Analysis
456.hmmer	35992	10.4	1	C	Sensitive Database Searching
458.sjeng	13487	5.8	14	C	A highly-ranked Chess Program
462.libquantum	4805	2	3	C	Qantum Computer Simulation running Shor's Algorithm
464.h264ref	51578	69.2	14	C	Video Compression Standard
471.omnetpp	48159	33.6	34	C++	Discrete Event Simulation of an Ethernet Network
473.astar	5842	2.2	3	C++	2D Path Finding Library used in Game AI
483.xalancbmk	326504	331	21	C,C++	XML Parser

Published SPEC performance numbers are a combination of two pairs of metrics, speed vs. rate and base vs. peak. Speed measures the time it takes to finish a single benchmark on the system, while rate is a throughput measure for how many parallel instances of a benchmark the system can handle in a certain time. Base and peak metrics refer to compiler options used to compile the benchmarks. Base is more restrictive and attempts to represent build options any application can use to get good compiler performance. For example, all benchmarks written in the same language must use the same compiler options. No feedback profiling is allowed. Peak does not have such restrictions. Any user visible compiler option can be used for any benchmark, and profile data on the train input set(s) may be generated. To be SPEC compliant each benchmark must be run 3 times on the reference input data set. Only SPEC compliant runs get accepted by the SPEC committee for publication on the SPEC website (http://www.spec.org). A full SPEC compliant run for a base speed publication for both CINT2006 and CFP2006 can take about 24 hours, depending on system configuration and compiler options.

Table 10 SPEC CPU2006 Floating-Point Benchmarks

Benchmark	LOC	CT	HR	LANG	Description
410.bwaves	918	4.6	2	Fortran	Computation of 3D Laminar Viscous Flow
416.gamess	932818	1649.4	10	Fortran	Atomic and Molecular Electronic Structure Analysis
433.milc	15042	154	5	C	Generator of Gauge Field with Dynamical Quarks
434.zeusmp	37326	74	8	Fortran	Astrophysical phenomena simulator
435.gromacs	87736	47.4	8	C	Newtonian Equation Solver
436.cactusADM	104047	40	1	Fortran	Einstein Equation Solver
437.leslie3d	3807	10.4	6	Fortran	Large-Eddy Simulations in 3D
444.namd	5315	9.2	9	C++	Simulation of Large Biomolecular Systems
447.dealII	199654	305	17	C++	Adaptive Finite Elements and Error Estimation
450.soplex	41417	25.4	14	C++	Linear Program Solver using the Simplex Algorithm
453.povray	157825	52	18	C++	Ray Tracer
454.calculix	49927	192.4	18	Fortran	Finite Element Solver
459.gemsFDTD	11580	40	4	Fortran90	3D Maxwell Equations Solver
465.tonto	143152	901.2	31	Fortran90	Quantum Chemistry Package
470.lbm	1176	1	1	C	Simulate Iincompressible Fluids in 3D
481.wrf2	217896	1101.2	33	Fortran90	Weather Research and Forcasting System
482.sphinx	207732	10.4	5	C	C Speech Recognition System

Data presented in this chapter are collected for a (non-compliant) single speed run using base compiler options. In base the compiler supports a wide variety of optimizations such as whole program optimizations, interprocedural optimizations like inlining, software prefetching, loop transformations, software pipelining, predication, global code scheduling and graph-coloring based register allocation. With base options no dynamic profile data may be used. Run-time performance data are normalized relative to the baseline. Performance and compile time data are relative to base options. Static data have been collected for implementations in an 11.1 compiler.

10.1 Dynamic and Static Results

The run-time benefits of multiple alloc and predicate-aware register allocation are in Table 11 for CINT2006 and in Table 12 for CFP2006. Multiple alloc is effective on CINT2006 with a gain of 1.9% on 400.perlbench and 1.18% on 458.sjeng. The gain overall in the geomean is 0.41%. These gains have to be seen in perspective. For example, on both 400.perlbench and 458.sjeng, the stall cycles due to RSE are about 5% (Desai et al. [26]). The 1.9% gain in 400.perlbench and the > 1% gain in 458.sjeng mean a more than 20% reduction in RSE stall cycles for these benchmarks. In case of 400.perlbench, all of them are due to the reduction of stall cycles in self-recursive function S_regmatch(), in which the benchmark spends more than 38% of its execution time.

Table 11 CINT2006 Performance Gains for MA and Predicate-aware (PA) Allocation

Benchmark	Multiple Alloc	PA w/ Strategy 1	PA w/ PQS	PA Delta
400.perlbench	1.90%	37.67%	37.67%	0.00%
401.bzip2	0.11%	5.01%	5.01%	0.00%
403.gcc	0.26%	1.71%	1.45%	-0.26%
429.mcf	0.00%	0.93%	0.93%	0.00%
445.gobmk	0.98%	2.91%	2.91%	0.00%
456.hmmer	0.00%	1.20%	1.20%	0.00%
458.sjeng	1.18%	8.01%	8.01%	0.00%
462.libquantum	0.55%	0.00%	0.37%	0.37%
464.h264ref	0.00%	0.00%	1.03%	1.03%
471.omnetpp	0.00%	0.40%	0.40%	0.00%
473.astar	0.00%	0.97%	0.00%	-0.96%
483.xalancbmk	0.00%	0.00%	0.00%	0.00%
Geomean	0.41%	4.48%	4.50%	0.01%

Table 12 CFP2006 Performance Gains for Predicate-aware (PA) Allocation

Benchmark	PA w/ Strategy 1	PA w/ PQS	PA Delta
410.bwaves	0.00%	0.00%	0.00%
416.gamess	8.74%	8.74%	0.00%
433.milc	0.00%	0.00%	0.00%
434.zeusmp	0.00%	0.00%	0.00%
435.gromacs	2.90%	2.90%	0.00%
436.cactusADM	1.03%	1.03%	0.00%
437.leslie3d	0.58%	0.58%	0.00%
444.namd	0.84%	0.84%	0.00%
447.dealII	0.00%	0.00%	-0.69%
450.soplex	0.23%	0.23%	0.23%
453.povray	1.07%	1.07%	0.11%
454.calculix	4.86%	4.86%	0.00%
459.GemsFDTD	0.84%	0.84%	0.83%
465.tonto	2.42%	2.42%	0.00%
470.lbm	0.65%	0.65%	0.32%
481.wrf	0.68%	0.68%	0.00%
482.sphinx3	1.36%	1.36%	0.00%
Geomean:	1.52%	1.52%	0.05%

Floating-point benchmarks did not show gains from multiple alloc and are not shown in Table 12. The gains from predicate-aware register allocation are for two different implementations, the PQS based implementation and the allocator that only models match and dominate live ranges precisely.

In Section 6.3 we classified four types of predicate live ranges: match, dominate, partition, and overlap and showed that simple live range tracking gives precise interference for match, dominate, as well as some partition and overlap live ranges. When a use predicate does not post-dominate all partition predicates or definitions overlap on many (=two or more) paths, complex live range tracking techniques must be employed for precise modeling of predicated live ranges. The implementation that tracks liveness under the qualifying use predicates and marks first predicate definitions gives practically identical run-time performance as the PQS based implementation. This suggests that relative simple predicate-awareness in the coloring allocator can reap the performance potential. The simple predicate-aware allocator models predicated live ranges precisely only for match and dominate live ranges. It is conservative for all partition and overlap live ranges. For the experiment, live ranges first were completed in the original control flow graph of the candidate region. Then a region-based reaching definition analysis was performed. Together with dominator information, this is sufficient to classify predicated live ranges in the region. When a live range falls into multiple classes, only the "most complex" class (overlap > partition > dominate > match) is accounted for. The data for predicated live ranges distribution are in Table 13 and Table 14. In CINT2006 there are only two benchmarks (402.bzip2 and 471.omnetpp) that have more than 10% (11.44% and 12.01%) partition and overlap live ranges. For all other benchmarks this number is below 10%. In CFP2006 five benchmarks (410.bwaves, 416.gamess, 433.milc, 447.dealII and 465.tonto) have more than 10% partition and overlap live ranges. For all benchmarks, the overlap live ranges usually account for less than 1%. The notable exceptions are 410.bwaves (4.99%), 401.bzip2 (2.61%) and 483.xalancbmk (2.39%). The data in the tables were collected for all predicated live ranges in all predicated regions of a benchmark. Since only relatively few overlap live ranges exist, a system like PQS or complex live range tracking is not necessary for precise predicated live range modeling. The data and code analysis suggests that the complex tracking cases are very rare. If conservative disjointness for partition and overlap live ranges is a concern, strategy 2 can model more than 95% of the predicated live ranges precisely for all benchmarks.

The run-time result for the predicate aware allocator is interesting for another reason also. In particular, the gains from the predicate-aware allocator can be higher than the gains from predication itself. The particular example is 400.perlbmk, where the gain from if-conversion overall is 1.95% (Table 15), but the gain from the predicate-aware allocator is greater than 37%. This can be explained as follows: when there is no predicate-aware allocator for predicated code, live ranges extend to outer loop boundaries. This increase in register pressure can only be avoided with a predicate-aware allocator. For example, the 37.67% gain in 400.perlbench is due to reducing RSE traffic in S_regmatch, the hottest (and self-recursive) function of the benchmark. Without the predicate-aware allocator all 96 register on the register stack get allocated. With the predicate-aware allocator only about half the number of registers are used. The increase in register pressure without the predicate-aware allocator can be explained with live range extensions in loops (Section 6.1). Another way to look at this is that the predicate-aware allocator must be present to secure gains from if-conversion. But, at least for the SPEC benchmarks, a simple predicate-aware allocator is sufficient to secure performance gains from if-conversion.

Table 13 Distribution of Predicated Live Ranges for CINT2006

Benchmark	match	dominate	partition	overlap
400.perlbench	73.92%	16.16%	9.46%	0.46%
401.bzip2	63.27%	22.68%	11.44%	2.61%
403.gcc	71.59%	20.78%	6.80%	0.83%
429.mcf	72.64%	20.46%	6.42%	0.48%
445.gobmk	77.94%	18.84%	2.91%	0.31%
456.hmmer	69.86%	22.67%	6.68%	0.79%
458.sjeng	73.68%	18.26%	7.37%	0.69%
462.libquantum	74.51%	16.67%	8.39%	0.44%
464.h264ref	74.70%	16.32%	8.87%	0.11%
471.omnetpp	66.77%	21.04%	12.01%	0.17%
473.astar	75.73%	18.63%	4.70%	0.94%
483.xalancbmk	69.81%	20.98%	6.82%	2.39%

Table 14 Distribution of Predicated Live Ranges for CFP2006

Benchmark	match	dominate	partition	overlap
410.bwaves	35.12%	48.24%	11.65%	4.99%
416.gamess	51.37%	36.90%	10.72%	1.01%
433.milc	48.36%	39.08%	11.99%	0.57%
434.zeusmp	63.94%	28.86%	6.15%	1.06%
435.gromacs	66.32%	24.39%	8.56%	0.72%
436.cactusADM	59.76%	30.35%	9.04%	0.84%
437.leslie3d	63.92%	26.51%	9.57%	0.00%
444.namd	64.15%	28.29%	7.56%	0.00%
447.dealII	68.70%	19.73%	11.42%	0.15%
450.soplex	75.06%	17.44%	6.74%	0.77%
453.povray	71.34%	21.76%	6.26%	0.64%
454.calculix	64.14%	28.44%	6.89%	0.52%
459.GemsFDTD	70.30%	20.06%	9.46%	0.18%
465.tonto	59.06%	30.11%	10.31%	0.52%
470.lbm	88.97%	10.29%	0.74%	0.00%
481.wrf	60.15%	31.10%	8.50%	0.25%
482.sphinx3	58.93%	33.30%	7.66%	0.11%

Table 15 Performance Gains from Predication for SPEC CPU2006

CINT2006	Predication Gains
400.perlbench	1.95%
401.bzip2	1.40%
403.gcc	1.18%
429.mcf	2.83%
445.gobmk	9.62%
456.hmmer	0.40%
458.sjeng	6.04%
462.libquantum	0.18%
464.h264ref	4.30%
471.omnetpp	0.20%
473.astar	26.67%
483.xalancbmk	1.70%
Geomean	4.49%

CFP2006	Predication Gains
410.bwaves	2.94%
416.gamess	6.23%
433.milc	0.78%
434.zeusmp	0.00%
435.gromacs	2.90%
436.cactusADM	0.69%
437.leslie3d	0.00%
444.namd	0.00%
447.dealII	2.86%
450.soplex	0.92%
453.povray	8.36%
454.calculix	7.86%
459.GemsFDTD	1.69%
465.tonto	1.60%
470.lbm	5.84%
481.wrf	0.00%
482.sphinx3	3.72%
Geomean:	3.02%

The advanced UNAT algorithm is effective and statically delivers the expected result by removing all spills of the ar.unat register. The data in Table 16 show that the new algorithm removes all ar.unat spills which the simple and conservative algorithm inserts.

On SPEC run-time gains from this optimization did not materialize, because the redundant ar.unat spills were only in cold code. Most of the ar.unat references were in loop intensive functions, where the conservative NaT propagation algorithm propagated the NaT property outside the loops or into cold outer loops. In cold code the extra ar.unat spills don't hurt. The data in Table 16 show that the static number of removed ar.unat references for CINT2006 is significant in 403. gcc, 458.sjeng, and 483.xalancbmk. For CFP2006, 416.gamess, 435.gromacs, 453.povray and 481.wrf benefit from the advanced UNAT algorithm.

Table 16 Effectiveness of advanced UNAT Algorithm

CINT2006	#spills/fills saved
400.perlbench	0
401.bzip2	0
403.gcc	35503
429.mcf	0
445.gobmk	889
456.hmmer	0
458.sjeng	90
462.libquantum	0
464.h264ref	1642
471.omnetpp	0
473.astar	0
483.xalancbmk	2664

CFP2006	#spills/fills saved
410.bwaves	0
416.gamess	11698
433.milc	0
434.zeusmp	0
435.gromacs	141
436.cactusADM	0
437.leslie3d	0
444.namd	0
447.dealII	0
450.soplex	0
453.povray	0
454.calculix	0
465.tonto	0
470.lbm	0
481.wrf	21926
482.sphinx3	0

10.2 Compile Time Data

The code generator in the Itanium compiler has major optimization phases like software pipelining, if-conversion, (global) code scheduling and register allocation. Since it also supports many classical optimizations it is an optimizing compiler back-end. The coloring allocator is usually not the top compile time consumer, although the coloring allocator has to manage aggressively optimized, predicated, pipelined and speculated code. For comparison, Table 17 and Table 18 show the relative compile time spent in the code generator overall, software pipeline, code scheduler, and register allocator.

Table 17 Compile Time Distribution of Code Generator Phases in CINT2006

Benchmark	Code Generation	Software Pipelining	Scheduling	Register Allocation
400.perlbench	38%	0%	17%	9%
401.bzip2	38%	7%	15%	8%
403.gcc	42%	5%	18%	7%
429.mcf	43%	9%	19%	6%
445.gobmk	41%	6%	17%	6%
456.hmmer	44%	10%	16%	7%
458.sjeng	36%	2%	15%	8%
462.libquantum	43%	9%	16%	6%
464.h264ref	39%	16%	11%	5%
471.omnetpp	31%	0%	11%	6%
473.astar	45%	15%	18%	4%
483.xalancbmk	33%	1%	15%	6%

Table 18 Compile Time Distribution of Code Generator Phases in CFP2006

Benchmark	Code Generation	Software Pipelining	Scheduling	Register Allocation
410.bwaves	37%	11%	11%	7%
416.gamess	31%	10%	5%	12%
433.milc	14%	7%	5%	1%
434.zeusmp	35%	12%	12%	5%
435.gromacs	37%	8%	15%	6%
436.cactusADM	36%	7%	14%	6%
437.leslie3d	30%	2%	10%	9%
444.namd	29%	7%	9%	6%
447.dealII	29%	5%	11%	3%
450.soplex	34%	5%	15%	4%
453.povray	34%	9%	14%	3%
454.calculix	24%	4%	8%	5%
459.GemsFDTD	25%	2%	7%	6%
465.tonto	25%	2%	10%	5%
470.lbm	41%	26%	9%	1%
481.wrf	35%	4%	18%	7%
482.sphinx3	42%	11%	15%	6%

The compile times are measured in triple runs using the compile time measurement capability in the compiler, which can be activated with the –i_tapi option. For example, 42% of the total compile time for 403.gcc is spent in the code generator, 5% in the pipeliner, 18% in the scheduler and 7% in the allocator. 12%   of the code generator compile time spent is spent in other phases including the if-converter.

The tables show that in practice the register allocator in the Itanium compiler is not a compile time bottleneck. In general the time spent in the allocator is well below 10%, with the exception of 416.gamess, where the allocator spends 12%.

10.2.1 Cost of Predication-Aware Allocation

Table 19 and Table 20 show the compile time cost for predicate-aware register allocation. The base is the predicate-unaware allocator: it does not model predicate live ranges for global variables. In particular it conservatively assumes any two predicated live ranges interfere and no predicated definition of a global live range is the start of the live range, unless the definition is in acyclic code. In cyclic code or loops, the start of live range is recognized implicitly at the pre-header of the outmost loop of the loop nest that contains the definitions. This is driven by the approximation of a live range as the intersection of a forward available variable analysis and backward live variable analysis (see Chapter 3.2, p. 37). On average the cost of predicate-aware allocation is about 46% for CINT2006 and 25% for CFP2006 for the match-or-dominate strategy. A PQS based predicate-aware allocator is-compared to the match-or-dominate allocator-5% on average slower on CINT2006 and 12% on average slower on CFP2006. The match-or-dominate allocator models match and dominate live ranges precisely, but is conservative for partition and overlap live ranges. Conservative means that the match-or-dominate allocator may infer false interferences between two live ranges. In other words, it may assume that two live ranges interfere although they do not. The extra interferences can sometimes hurt overall compile-time. For example, register assignment time can increase and there can be extra reconciliation become necessary at region boundaries. Very few benchmarks show behavior, and the only relevant example is the 4.2% slowdown for 464.h264ref. Ignoring compile time differences of less than 1% than for all other benchmarks PQS based is more compile-time expensive than match-or-dominate without-at least for the given if-converter-giving performance benefits (cf. Table 11 and Table 12 for performance). It is curious that the predicate-aware allocator can actually speed-up allocation time. The two major examples for this phenomenon are 416.gamess and 440.namd in CFP2006 (Table 20), where predicate-aware allocation improves allocation time by at least 6% and up to 18.89%. The reason is that the predicate-unaware allocator "sees" more interference and needs more allocation iterations (Chapter 3).

Table 19 CINT2006 - Compile Time Cost of Predicate-Aware Register Allocation

Benchmark	PA w/ PQS	PA w/ Strategy 1 (S-1)	PQS vs. S-1
400.perlbench	64.26%	63.99%	0.28%
401.bzip2	55.94%	50.90%	5.04%
403.gcc	107.02%	98.22%	8.80%
429.mcf	-2.56%	-2.71%	0.14%
445.gobmk	1.74%	2.30%	-0.56%
456.hmmer	42.67%	40.83%	1.84%
458.sjeng	31.80%	17.88%	13.92%
462.libquantum	55.20%	45.60%	9.60%
464.h264ref	2.35%	6.55%	-4.20%
471.omnetpp	135.03%	129.73%	5.31%
473.astar	52.41%	45.94%	6.47%
483.xalancbmk	66.40%	52.27%	14.12%

Base: Predicate-Unaware Register Allocator

Table 20 CFP2006 - Compile Time Increase of Predicate-Aware Register Allocation

Benchmark	PA w/ PQS	PA w/ Strategy 1 (S-1)	PQS vs. S-1
410.bwaves	14.48%	0.27%	14.20%
416.gamess	-6.24%	-10.88%	4.65%
433.milc	27.46%	16.16%	11.30%
434.zeusmp	7.63%	3.95%	3.68%
435.gromacs	38.66%	17.67%	20.99%
436.cactusADM	99.70%	88.91%	10.79%
437.leslie3d	16.14%	10.24%	5.90%
444.namd	-17.46%	-18.89%	1.43%
447.dealII	62.51%	41.32%	21.19%
450.soplex	45.01%	28.61%	16.40%
453.povray	125.18%	104.59%	20.59%
454.calculix	12.74%	10.90%	1.85%
459.GemsFDTD	9.77%	-3.54%	13.31%
465.tonto	7.27%	5.35%	1.92%
470.lbm	34.75%	27.12%	7.63%
481.wrf	16.40%	5.29%	11.11%
482.sphinx3	37.84%	29.00%	8.84%

Base: Predicate-Unaware Register Allocator

10.2.2 Cost of Speculation-Aware Allocation

In Chapter 7 we discussed speculation-aware allocation. For control-speculation, we showed that under the assumption that the destination of a speculative load (ld.s) and the source of its chk.s match (=are the same symbolic or virtual register), an effective algorithm exists that avoids all spills of the ar.unat register. The allocator must be speculation-aware to guarantee program correctness. Table 21 shows the cost of control-speculation awareness. For the advanced NaT propagation algorithm (Section 7.1.2), the average allocation cost is 10.26% for CINT2006 and 7.3% for CFP2006. Data for control-speculation were measured by turning speculation support off in the allocator. Since allocation time in general is less than 10% of the overall compile time (see Table 17 and Table 18), speculation support itself consumes less than 1%.

Table 21 Compile Time (CT) Cost of Control Speculation

CINT2006	CT cost of speculation
400.perlbench	7.60%
401.bzip2	6.50%
403.gcc	8.42%
429.mcf	24.76%
445.gobmk	9.75%
456.hmmer	9.93%
458.sjeng	7.35%
462.libquantum	10.74%
464.h264ref	5.95%
471.omnetpp	7.90%
473.astar	13.60%
483.xalancbmk	10.57%
Average	10.26%

CFP2006	CT cost of speculation
410.bwaves	3.38%
416.gamess	3.29%
433.milc	10.32%
434.zeusmp	6.49%
435.gromacs	7.59%
436.cactusADM	6.94%
437.leslie3d	3.46%
444.namd	8.18%
447.dealII	9.57%
450.soplex	10.55%
453.povray	10.75%
454.calculix	6.54%
459.GemsFDTD	5.50%
465.tonto	7.50%
470.lbm	8.00%
481.wrf	5.02%
482.sphinx3	10.94%
Average	7.30%

10.2.3 The Case for Scalable Register Allocation

Since region-based allocation methods require global structure, for example, for region reconciliation and in data flow analysis, this thesis proposed a scalable approach that partitions symbolic registers and possibly the register file(s). Figure 49 shows the normalize compile times for a serial scalable allocator. The (non-standard) test care for evaluation is “f_serverapp”, which is a generated function in major industry server application with > 500 K register candidates. The x-axis in Figure 49 shows the number of partitions. The y-axis shows normalized compile times, where the time for eight partitions is one. For the measurements, the global symbolic registers (candidates) were divided into equal 2, 4, 8, 16 and 32 partitions of equal subsets. For each subset the allocator is run. The output of allocation N is the input to allocation N+1. For example, the input to the second run is the symbolic registers of the second partition and the physical registers the candidates of the first partition got assigned. The number of partitions determines the number of allocations.

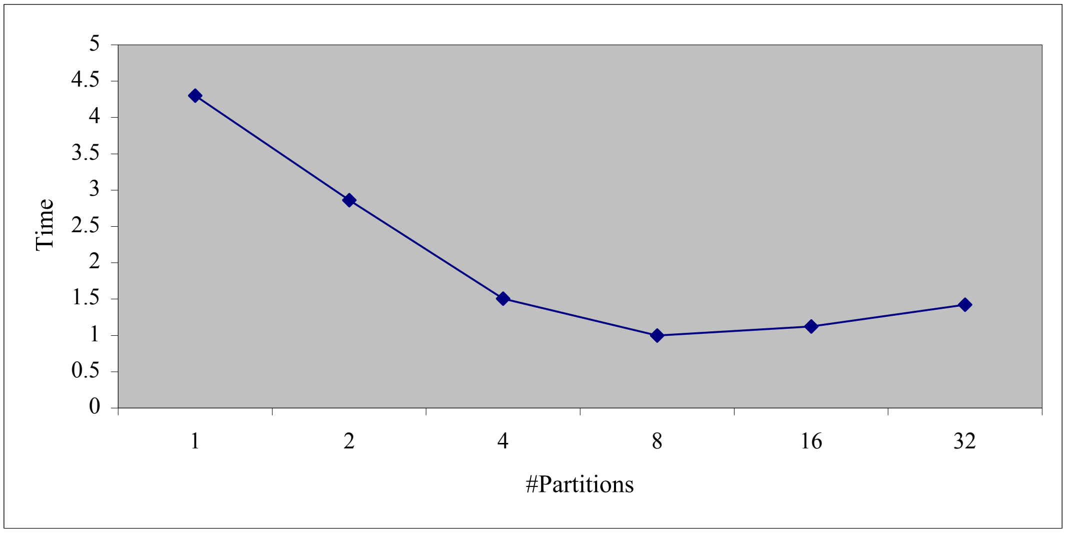


Figure 49 Compile Time for Serial Scalable Allocator

The graph of Figure 49 is close to a line up to 8 partitions. Let CT(2^k) be the compile time for a partition. Then CT(2^k) =($\sqrt{2}$)$^{-1}$ CT(2^{k-1}) for k=1,2,3 gives the approximate relation between the partitions. For more than 8 partitions (for 16 and 32) partitions compile time increases. In these cases allocation cost outweighs the savings from partitioning.

The parallel scalable allocator also partitions the register file accordingly. For example, when there are 8 partitions of the register candidates, the available (physical) registers are also partitioned by a factor of 8. In the case of a parallel scalable allocator, the compile time must decrease with the number of partitions. The x-axis in Figure 50 shows the number of partitions. The y-axis shows normalized compile times, where the time for 32 partitions is one. The number (>150) for one partition is not shown in the graph. The numbers for the parallel allocator are experimental and optimistic. For each partition scheme the compile-time was measured for one specific partition (both candidates and machine registers) and then extrapolated. Also, the synchronization overhead in the case of spill code is not taken into account. Since candidates in different partitions may interfere (although they are assumed they do not) they cannot share the same spill location. That they cannot share the same register is guaranteed by partitioning the register file(s). The small number of registers available per partition does not have a material compile time impact in our experiment.

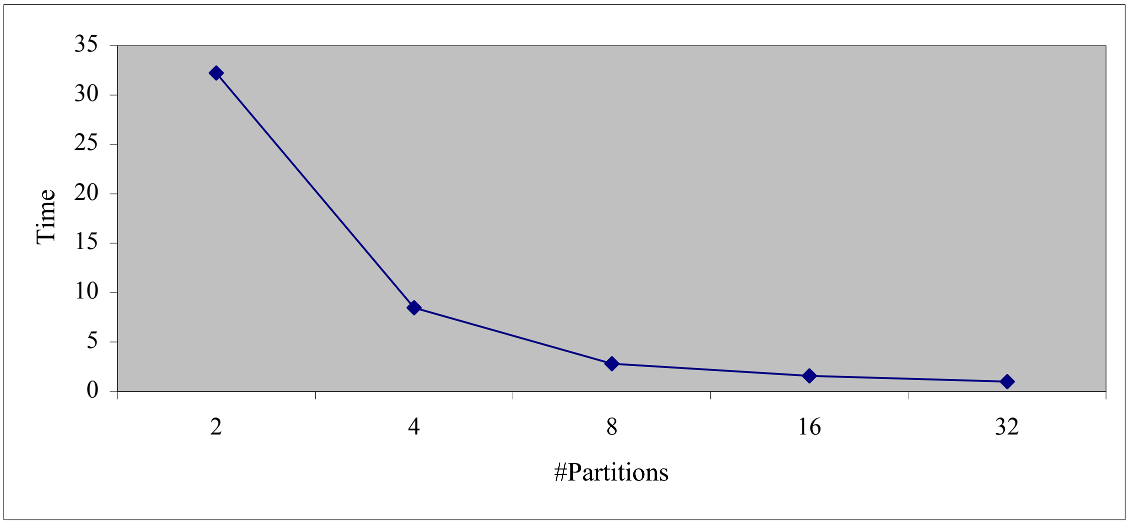


Figure 50 Compile Time for Parallel Scalable Allocator

While the compile data for scalable allocators are preliminary, they show the potential of the technique. The “best” configuration of a scalable allocator depends on the allocation problem. There is no one-size-fits-all solution. For example, for our “golden” test case, 8 partitions are the compile time sweet spot (Figure 49), but for allocation problems with half the number of candidates, 4 partitions are likely to be the better choice.

11 Conclusions and Future Work

This thesis described extensions of a coloring allocator covering features provided by the Itanium architecture like predicated code, control- and data speculation and dynamic register stack.

Predicated code: It classified predicated live ranges and showed that classical techniques can be used effectively to engineer efficient coloring allocators for predicated code. Specifically, when predicated code is generated from compiler control flow expensive predicate analysis frameworks, like PQS, don't have to be employed.

Speculated code: It described a new method of efficiently allocating speculated live ranges avoiding spill code compared to a conservative method. The new method solves the NaT propagation problem efficiently.

Dynamic register stack: It presented methods to dynamically control the register stack effectively in particular for regions with function calls and/or pipelined loops.

Scalable allocation: It proposed the scalable allocator as a generic coloring method capable of allocating effectively programs with a large set of register candidates. It demonstrated that this method can be used also for parallel allocation e.g. on multi-core machines.

Despite the rich body of work in the field of register allocation future work is plenty, in the context of this thesis and beyond. We conclude this thesis with a small selection:

1. The IA-64 register stack can be dynamically partitioned. In particular the (scratch) out registers on top of the stack are not limited to 8. In addition to partitioning schemes, more general allocation regions can be investigated.
2. The interface between pipeline-aware register allocation and a coloring allocator seems to be mostly neglected in literature. Instead, pipeline allocators and coloring allocators are treated separately. On IA-64 rotating registers are a scarce resource and the pipeline allocator leaves candidates for the coloring allocator. The impact of pipeline-awareness and the potential benefits in a coloring allocator have not been investigated or published.

3. The classification of predicate live ranges in Chapter 6 is intuitive and the proofs of the theorems in Section 6.3 are informal. A formal theory that derives the classification and proofs of the theorems would put the results on a stronger mathematical foundation.
4. The scalable allocator is at a conceptual state. More experiments need to be conducted to explore the concept and configurations, especially with respect to the cost and benefits of parallel allocations.
5. There is no compiler framework that includes and evaluates the merits of the various ideas for coloring allocators. Results are reported in different experimental environments. This makes results difficult to compare and judge relative merits of methods proposed.
6. There seems to be no study that compares optimal methods with best of class coloring allocators and linear scan allocators on a wide class of benchmarks.

12 Appendix

12.1 Assembly Code Example

```
          Instruction                          Cycle
 { .mmi
        alloc   r14=ar.pfs,1,8,0,8    //0:
        add     r8=1,r0               //0:  [1]
        mov     r11=ar.lc             //0:
 } {   .mib
        cmp4.eq.unc     p7,p0=r32,r0  //1:  [2]
        mov     r40=pr                //1:
(p7)   br.cond.dpnt    .b1_3 ;;       //1:  [2]
// Block 1:  Pred: 0     Succ:
 }{.mii
        nop.m   0
        mov     ar.ec=1               //0:  [3]
        zxt4    r3=r32                //0:
 }{.mii
        mov     r33=1                 //0:
        mov     pr.rot=0x10000        //0:  [4]
        add     r2=-1,r3 ;;           //0:
 }{.mib
        nop.m   0
        mov     ar.lc=r2              //1:  [5]
        nop.b   0 ;;
 }
.b1_4: // Block 4:pipelined  Pred:1 4  Succ:4 3
 { .mmi
 (p16)  setf.sig  f32=r8              //0:
 (p16)  setf.sig  f33=r33             //0:
        nop.i   0 ;;
 }{.mfi
        nop.m   0
 (p16)  xma.l   f34=f32,f33,f0        //6:
        nop.i   0 ;;
 }{.m_mi
 (p16)  getf.sig  r8=f34 ;;           //10: [6]
 (p16)  add     r32=1,r33             //14:
        nop.i   0
 }{.mib
        nop.m   0
        nop.i   0
        br.ctop.sptk    .b1_4 ;;      //14:
 }
// Block 3: exit epilog modified  Pred: 1 4
.b1_3:
 { .mi_i
        nop.m   0
        mov     ar.lc=r11 ;;          //0:
        mov     pr=r40,0x1003e        //1:
 }{.mib
        nop.m   0
        nop.i   0
        br.ret.sptk.many        b0 ;; //1:  [2]
}
```

Code and Comments

```
[1] res = 1;

[2] if (n==0)return res;

Initializations:

[3] epilog count(ec):
    ec=1

[4] Set first rotating
    stage predicate
    p16=1
[5] Setloop count(lc)
```

```
Pipelined loop:

for (i=1; i<=n; i++) {
    res = res * i;
}

[6] r8 is result and
    return register.
```

```
[2] return res; // in r8
```

Figure 51 Itanium Assembly for Faculty Function

12.2 Edge Classification, Irreducibility and Disjointness

This section lists a few basic facts about graphs that are not commonly found in the literature.

Let G(V,E) be a directed graph. The edges E can be partitioned as follows:

Let PRE(n) be the preorder number and RPO(n) the reverse post-order number. Let $h \rightarrow t$ be an edge from node h to t. Then the following edge partition is the result of a depth-first traversal (*dft*):

- If (PRE(h) < PRE(t)) then $h \rightarrow t$ is either a tree or forward ("advancing") edge.
- If (PRE(h) > PRE(t)) then $h \rightarrow t$ is a cross edge .
- If (RPO(h) > RPO(t)) then $h \rightarrow t$ is a retreat edge.

An edge $h \rightarrow t$ is a back edge if h *dominates* t.

A special case of a back edge is when h identical with t (and trivially PRE(h)==PRE(t), RPO(h)==RPO(t)).

Edge partition is not absolute, but relative to the depth-first search (dfs). For example, a tree edge in one traversal could be a cross edge in another traversal. Or a retreat edge in one traversal could become a forward edge in another dfs. The reason is that the edge selection during dfs is non-deterministic, and different selections of edges change node numbering and edge classification (=partitioning).

A graph is reducible if all retreat edges are back edges. Back edges are invariant retreat edges: *any* depth-first search recognizes them as retreat edges. If there is a retreat edge that is not a back edge the graph is irreducible.

Irreducibility can be recognized indirectly, but simply: If the classical dataflow algorithm for dominator calculation does not terminate in two iterations (the first iteration to compute the dominator tree and the second iteration to check the termination), the graph is irreducible.

In case of irreducible graphs global disjointness, which is computed on the acyclic graph that has retreat edges removed, may not match run-time disjointness. Figure 52 demonstrates this based on two depth-first traversals:

In depth-first traversal B1 $\rightarrow$ B2 $\rightarrow$ B3 $\rightarrow$ B4 $\rightarrow$ B5 $\rightarrow$ B6 the reverse post-order numbers for the blocks are: (B1|1), (B2|2), (B3|3), (B4|4), (B5|5) and (B6|6). The edge from B5 to B3 is recognized as a retreat edge. Removing the retreat edge and computing disjointness will yield that B2 and B6 are not disjoint. Since B5 does not dominate B3, the retreat edge is not a back edge.

In depth-first traversal B1 → B4 → B5 → B3 → B6 → B2 the reverse post-order numbers for the block are: (B1|1), (B2|2), (B3|6), (B4|4), (B5|5) and (B6|6). The edge from B3 to B4 is recognized as a retreat edge. Removing the retreat edge and computing disjointness will yields that B2 and B6 are disjoint. Since B3 does not dominate B4, the retreat edge is not a back edge.

Gillies et al. [31] suggests there can be "inaccuracies" for global predicate relations, but in fact there can be a fundamental stability problem. The problem is that global disjointness computed by compiler may not match run-time disjointness. This mis-match could result in register overwrites. For example, if B4, B5 and B6 are if-converted, and the retreat edge is $B5 \rightarrow B3$, a local live range in B5 could overwrite a global live range with a use in B3. On the other hand, removing only back edges, but not retreat edges, gives conservative global disjointness information since cycles remained in the graph.

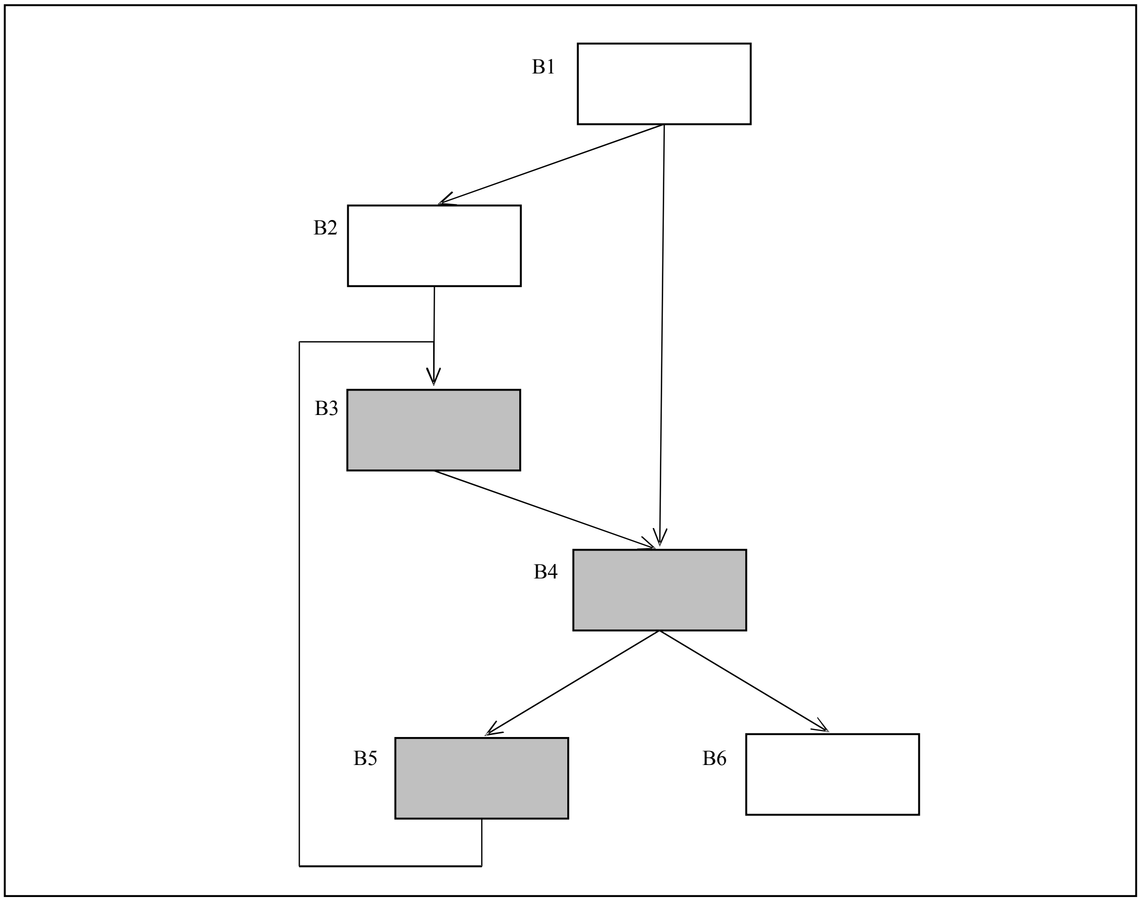


Figure 52 Example for an Irreducible Graph

12.3 PQS Queries

In PQS a block predicate P also represents its execution set. The execution set is the set of traces for which P is set (=true). In this interpretation we can speak of subsets of predicates and partitions of predicates. It is the basis of the predicate partition graph (PPG), which is used by PQS queries to derive predicate relations. The key queries are lub_sum(P,E) and lub_diff(P,E), which are used in predicate-aware dataflow analysis routines and interference graph construction. The least upper bound sum (lub_sum(P,E) in Figure 53) gives the smallest superset for the execution sets represented by the union of predicate P and the set of predicates, E. The resultant set is reduced to parent nodes in the PPG for each complete partition that contains P. One application of the least upper bound sum is to determine the set of predicates under which a candidate is live, if it is live under Q and P. The approximation is necessary since the predicates available may not be sufficient to express the union of predicate sets accurately in all cases.

```
Set reduce(Predicate P, Set E)
// if (R = P|Qi and all Qi ∊ E, replace P|Qi by R)

Set lub_sum(Predicate P, Set E)
      Set E’= {}; // empty set
      foreach Q in E
            if (Q ⊂ P)
                  continue;
            elif (P ⊆ Q)
                  return E;
            else
                  E’ = E’ + Q;
            fi
      endfor
      return (reduce(P, E’));
```

Figure 53 PQS Query Least Upper Bound Sum

The least upper bound difference (lub_diff(P,E) in Figure 53) computes the smallest superset for the execution set represented by the set difference of predicate set E and predicate P. One application of the least upper bound difference is to determine the set of predicates under which a candidate is live, if it is live under Q and killed under P. The approximation is necessary since the predicates available may not be sufficient to express the difference of predicate sets accurately in all cases. Intuitively the least upper bound difference computes representatives of all paths where a candidate could still be live after it is killed under predicate P.

```
Set rel_cmpl(Predicate P, Predicate Q)
     if (!is_subset(P,Q)) return {}; // empty set.
     E’ = {}; // empty set
     foreach path from Q to P
          foreach edge R→S
               foreach partition R→S|T
                    E’=E’+T
               endfor
          endfor
     endfor
     return E’;

Set approx_diff(Predicate P, Predicate Q)
     // find_lca(P,Q): “least common ancestor of P and Q”
     return rel_cmpl(P, find_lca(P,Q));

Set lub_diff (Predicate P, Set E)
     Set E’= {}; // empty set
     foreach Q in E
          if (Q ⊂ P)
               continue;
          elif (P ⊆ Q)
               E’=E’+rel_cmpl(P,Q);
          elif (is_disjoint(P,Q))
               E’=E’+Q;
          else
               E’ = E’+approx_diff(P,Q);
          fi
     endfor
     return (reduce(P, E’));
```

Figure 54 PQS Query Least Upper Bound Difference

13 List of Figures

14 List of Examples

15 List of Tables

16 List of Theorems

17 Glossary

Symbols

K	number of colors (=number of machine registers)
V1,V2, ….	global register candidates
v1, v2, …	local register candiates
A, B, C, …	global variables

Definitions

Application

A program. For example, a SPEC benchmark is an application

Available variable

A variable is available at a point when there is at least one path from program entry to the point

Backedge, back edge

Edge $t \rightarrow h$ in directed graph, where h dominates t

Chordal Graph, chordal

In every cycle of length >= 4 there is a chord. A chord is an edge to a cycle node that is not an immediate neighbor. Efficient graph-coloring algorithm is known when interference graph is chordal. Chordal graphs are subsets of *perfect graphs*.

Chromatic Number

Exact number of colors needed to color a graph. Determining the chromatic number for an arbitrary graph is NP complete.

Control flow graph

A directed graph.

Clique

A complete subgraph. All nodes in the subgraph are pairwise adjacent ("mutally connected").

Clique Number

Order of largest clique in a graph

Completion

The insertion of a (empty) basic block on a *critical edge*.

Constrained Node

Node n with degree(n) >= K

Control speculation ("*breaking the branch barrier"*)

Compiler optimization that hoists a chain of instructions starting at a load above one or more controlling branches

Critical Edge

A edge from basic block A to B is critical is A has two or more successors and B has two or more predecessors

Data speculation ("*breaking the store barrier"*)

Compiler optimization that hoists a chain of instructions starting at a load above one or more (possibly dependent) stores

Degree of a node n (=degree(n))

Number of edges of a node in an undirected graph

Disjoint predicates

Set of predicates. No two predicates in the set can be 'true' at any given program point

Dominance

Node A dominates node B if all paths from the single entry block to B contain A. In this case A is said to *dominate* B.

END block

Last node in a control flow graph. Every control flow graph can be transformed to have a unique END node.

End of live range

Last use of a symbolic register on any path from *START* to *END.* Note that some authors consider the last use as the start of the live range e.g. Gillies et al. [31].

EPIC

Explicitly parallel instruction computing

Execution Set

Set of *execution traces*.

Execution Trace

Set of predicated instructions in a predicated region.

Hyperblock

Predicated *superblock.*

ILP

Instruction level parallelism

Interference graph

Undirected graph G(V, E). The set V of nodes represents register candidates. There is an edge from node A to node B if the nodes cannot be assigned the same register ("node A and B interfere")

Irreducible graph

Control flow graph that is not *reducible*.

JS block

Join-Split block. Basic block inserted on *critical edge*. Purpose: improves accuracy of predicate analysis (associated with JS block is a block predicate)

LP

Linear Programming

Live range

Set of program points where a variable is live and available

Live variable

A variable is live at a point when there is at least one path from the point to a program exit that contains a use of the variable

NaT bit

"Not a Thing": Extra bit in general register that signed a speculation fault or exception

NaT consumption fault

Exception or fault that occurs because the NaT bit is set unexpectedly

NaT producer

A speculative load instruction (ld.s)

NP-noise

Observation that local assignment can change global allocation outcome ("chaotic behavior")

*Partition (*of predicates*)*

Set of disjoint predicates

Path

Sequence of instructions or basic blocks in a control flow graph

Perfect Graph

Graphs where cyclic number equals chromatic number

Predicate Set

Set of predicates under which a *live range* is live. This is non-standard terminology.

Predicate Partition Graph (PPG)

Directed acyclic graph whose nodes are predicates and whose labeled edges represent partition relations between predicates. PPG queries interpret predicates are interpreted as *execution sets.* A PPG is complete if in contains a unique root node from which all nodes are reachable.

Predication

Conditional execution of an instruction guarded by a qualifying predicate

Pre-materialization

Shrinks re-computable live ranges *before* register allocation

Program

Synonym for function or procedure

Reconciliation code

A live range may be assigned different memory location (e.g. two different registers or a register and memory). Instructions that map between different assignements (e.g. a move to reconcile different register assignments) constitute reconciliation code

Reducible graph

Control flow graph that has no retreat edge ("acyclic graph") or all retreat edges are back edges.

RSE

Register Stack Engine. A unit on the Itanium processor managing dynamic register stacks.

Significant node

Interference graph node with more than K edges

Simplification phase

Phase in graph coloring register allocator that maps the nodes in the interference graph onto the coloring stack. It uses the simlification criterion to remove unconstrained nodes from the interference graph. When no unconstrained nodes can be found, it uses spilling heuristics.

Simplification criterion (classical version):

remove unconstrained nodes from interference graph and push it on the coloring stack.

Spilling

1. Allocating memory to a symbolic register
2. Removal of node from interference graph

Speculation

Early execution of an instruction

START block

First node in a control flow graph. Every control flow graph can be transformed to have a unique START node.

Start of live range

First definition of a symbolic register on any path from *START* to *END.* The definition can implicit and determined by data flow analysis as the first point of any path from *START* to *END* where the symbolic register is both live and available. Note that some authors consider the first definition to be end of a live range, for example Gillies et al. [31].

Strict program

Let V be any variable. For each path from program entry to a use of V there is a definition of V.

Superblock

Control flow graph structure which has single entry and multiple exits. This is one generalization of a basic block.

Unconstrained node

Node n with degree(n) < K

18 References

[1] A. V. Aho, M.S. Lam, R. Sethi, and J. D. Ullman, "Compilers: Principles, Techniques, & Tools", Addison Wesley, Second Edition 2007

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren, "Conversion of Control Dependence To Data Dependence", in Proceedings of the 10th ACM Symposium on Principle of Programming Languages, POPL'83, January 1983, pp. 177-189

[3] A.W. Appel, "Modern Compiler Implementation in Java", Cambridge University Press, First Edition 1997

[4] A.W. Appel, and L. George, "Optimal Spilling for CISC Machines with Few Registers", Programming Language Design and Implementation (PLDI), June 2001, pp. 243-253

[5] I.D. Baev, R.E. Hank, and D.H. Gross, "Prematerialization: reducing register pressure for free", 15th International Conference on Parallel Architecture and Compilation Techniques (PACT 2006), pp. 285-294

[6] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe, "Spill code minimization via interference region spilling", Programming Language Design and Implementation (PLDI), June 1997, pp. 287-295

[7] D. Bernstein, D. Q. Goldin, M.C. Golumbic, H. Krawczyk, Y. Mansour, I. Nashon, and R.Y Pinter "Spill code minimization techniques for optimizing compilers", Proceedings of the ACM SIGPLAN '89 Programming Language Design and Implementation (PLDI), July 1989, pp. 258-263

[8] J. Bharadwaj, W. Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce, "The Intel IA-64 Compiler Code Generator", IEEE Micro, Sep./Oct. 2000, pp.44-52

[9] J. Bharadwaj, K. Menezes, and C. McKinsey, "Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs", Proceedings of the 33rd International Symposium on Microarchitecture, MICRO-33, December 2000 , pp. 262-271

[10] F.Bouchez, A. Darte, and F. Rastello, "On the Complexity of Register Allocation", IEEE International Symposium on Code Generation and Optimization (CGO'07), March 2007, pp. 102-114

[11] P.Briggs, K.D.Cooper, L. Torczon,"Rematerialization", Proceedings of the ACM SIGPLAN '92 Language Design and Implementation (PLDI), June 1992, pp. 311-321

[12] P. Briggs, "Register Allocation via Graph Coloring", PhD thesis, Rice University, April 1992

[13] P. Briggs, K.D.Cooper, and L.Torczon. "Improvements to Graph Coloring Register Allocation", ACM Transactions on Programming Languages and Systems, Vol. 16, No. 3, May 1994, pp. 428-455

[14] P. P. Briggs, K.D.Cooper, K. W. Kennedy, and L.Torczon. "Digital computer register allocation and code spilling using interference graph coloring", US Patent 5,249,295 (1993)

[15] D.Callahan, and B. Koblenz, "Register Allocation via Hierarchical Graph Coloring", Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991, pp. 192-203

[16] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. "Register allocation via coloring", Comp. Lang. 6 (1) , 1981, pp.47-57

[17] G. J. Chaitin. "Register Allocation and Spilling via Graph Coloring", Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, 1982, pp. 98-105

[18] G.J. Chaitin, “Register Allocation and Spilling via Graph Coloring”, US Patent 4,571,678 (1986)

[19] Y. Choi, A.D. Knies, L. Gerke, T-F. Ngai, “The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor”, Proceedings of the 34th International Symposium on Microarchitecture, MICRO-34, December 2001, pp. 182-191

[20] F.C. Chow, “A Portable Machine-Independent Global Optimizer – Design and Measurements”, PhD Thesis, Stanford University, 1983, pp. 70-87

[21] F.C. Chow, "Minimizing Register Usage Penalty at Procedure Calls", Proceedings of the ACM SIGPLAN ’88 Conference on Programming Language Design and Implementation (PLDI), June 1988, pp. 85-95

[22] F.C. Chow and J.L. Hennessy. “The priority-based coloring approach to register allocation”, ACM Trans. On Programming Languages and Systems 12, No. 4 , 1990, pp. 501-536

[23] K.D. Cooper, A. Dasgupta, and J. Eckhardt, “Revisiting Graph Coloring Register Allocation: A Study of the Chaitin-Briggs and Callahan-Koblenz Algorithms”, LCPC 2005, LNCS 4339 2007, pp. 1-16

[24] K. D. Cooper and L. Taylor Simpson, “Live Range Splitting in a Graph Coloring Register Allocator”, Proceedings of the 7th International Conference on Compiler Construction CC’1998, LNCS 1383, 1998, pp. 174-187

[25] K.D. Cooper, A. Dasgupta, and J. Eckhardt, “Revisiting Graph Coloring Register Allocation: A Study of the Chaitin-Briggs and Callahan-Koblenz Algorithms”, LCPC 2005, LNCS 4339, 2007, pp. 1-16

[26] D. Desai, G. Hoflehner, A. Kejariwal, D. M. Lavery, A. Nicolau, A. V. Veidenbaum, and C. McNairy, “Performance Characterization of Itanium 2 –Based Montecito Processor”, SPEC Benchmark Workshop 2009, pp. 36-56.

[27] A. Douillet, J. N. Amaral, G.R.Gao, “Fine-Grain Stacked Register Allocation for the Itanium Architecture “, 15th Workshop on Languages and Compilers for Parallel Computing (LCPC), 2002

[28] A. E. Eichenberger and E. S. Davidson, “Register allocation for predicated code,” in Proceedings of the 28th Annual International Symposium on Microarchitecture, MICRO-28, December 1995, pp. 180–191

[29] C. Fu and K.D. Wilken, “Optimal and near-optimal global register allocation using 0-1 integer programming”, Softw.Pract.Exper. 26(8), 1996, pp. 929-965

[30] L. George and A.W.Appel. “Iterated Register Coalescing”, 23rdACM Symposium on Principle of Programming Languages, POPL’96, 1996, pp. 208-218

[31] D. M. Gillies, R.D-C. Ju, R. Johnson, and M. S. Schlansker. “Global Predicate Analysis and its Application to Register Allocation”, Proceedings of the 29th International Symposium on Microarchitecture, MICRO-29, December 1996, pp. 114-125

[32] D. W. Goodwin and K.D. Wilken, ”A faster optimal register allocator”, Proceedings of the 35th International Symposium on Microarchitecture MICRO-35, November 2002, pp. 245-256

[33] R.Gupta, M.L.Soffa, and T. Steele. “Register Allocation Via Clique Separators”, Programming Language Design and Implementation (PLDI), June 1989, pp. 264-274

[34] S. Hack, D. Grund, and G. Goos, “Register Allocation for Programs in SSA-Form”, 15th International Conference on Compiler Construction CC’2006, LNCS 3923, 2006, pp. 247-262

[35] R. E. Hank, W -m. W. Hwu, B. R. Rau, “Region-based compilation: an introduction and motivation.”, Proceeding of the 25th Annual International Symposium on Microarchitecture MICRO-25, December 1992, pp. 158-168

[36] U. Hirnschrott, A. Krall, and B. Scholz, “Graph Coloring vs. Optimal Register Allocation for Optimizing Compilers”, Joint Modular Languages Conference, JMLC 2003, Proceedings. LNCS 2789, pp. 202-213

[37] G.Hoflehner and M. Davis, “Method and Apparatus for dynamic register scratching”, US Patent 7,647,482 (2010)

[38] G.Hoflehner and J. Pierce, “Method and Apparatus for inserting more than one alloc instruction within a routine”, US Patent 6,907,601 (2005)

[39] G. Hoflehner, K. Kirkegaard, R. Skinner, D. M. Lavery, Y-F. Lee, and W. Li, “Compiler Optimizations for Transaction Processing Workloads on Itanium® Linux Systems”, Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO-37), December 2004, pp. 294-303

[40] G. Hoflehner, “Strategies for Predicate-Aware Register Allocation”, CC 2010, LNCS 6011, pp. 185-204

[41] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, "Introducing The IA-64 Architecture", IEEE M icro, Sep/Oct 2000, pp. 12-22

[42] Intel Corporation, “Intel® Itanium® Architecture Software Developer’s Manual”, Vol. 1-3, Revision 2.2, January 2006, http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm

[43] Intel Corporation, "Intel® Itanium® 2 Processor Reference Manual”, May 2004, http://download.intel.com/design/Itanium2/manuals/25111003.pdf

[44] Intel Corporation, “Itanium™ Software Conventions and Runtime Architecture Guide”, May 2001, http://download.intel.com/design/itanium/downloads/245358.pdf

[45] R. Johnson and M. Schlansker, "Analysis techniques for predicated code", in Proceedings of the 29th International Symposium on Microarchitecture, MICRO-29, December 1996, pp. 100-113

[46] V. Kathail, M.S. Schlansker, and B.R. Rau, "HPL PlayDoh architecture specification: Version 1.0", Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, 1994 (Feb.)

[47] A.B. Kempe, “On the geographical problem of the four colors”, American Journal of Mathematics 2, 1879, pp. 193-200

[48] B.D. Koblenz and C.D. Callahan, “Register allocation methods having upward pass for determining and propagating variable usage information and downward pass for binding; both passes utilizing interference graphs via coloring”, US Patent 5,530,866 (1996)

[49] T. Kong and K.D. Wilken, “Precise register allocation for irregular architectures”, Proceeding of the 31st Annual International Symposium on Microarchitecture MICRO-31, December 1998, pp. 297-307

[50] A.Koseki, H.Komatsu, and T.Nakatani, “Preference-directed graph coloring”, Proceedings of the ACM SIGPLAN 1002 Conference on Programming Language Design and Implementation, June 2002, pp. 33-44

[51] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C. Lim, J. Ng, and D. Sehr, “An Advanced Optimizer for the IA-64 Architecture”, IEEE Micro 20(6), Nov/Dec. 2000, pp. 60-68

[52] M. S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation", June 1988, pp. 318-328.

[53] J. Lin, W -C. Hsu, P-C. Yew, R. D-C. Ju, and T-F. Ngai, "A Compiler Framework for Recovery Code Generation in General Speculative Optimizations," 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04), 2004, pp.17-28

[54] G-Y Lueh and T. Gross, "Call-Cost Directed Register Allocation", Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI), June 1997, pp. 296-306

[55] G-Y Lueh, T. Gross, and A-R Adl-Tabatabai, "Fusion-Based Register Allocation", ACM Transactions on Programming Languages and Systems, Vol. 22, No. 3, May 2000, pp. 431-470

[56] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann, "Effective compiler support for predicated execution using the hyperblock", Proceeding of the 25th Annual International Symposium on Microarchitecture MICRO-25, December 1992, pp. 45-54

[57] S. A. Mahlke, R. E. Hank, R. A. Bringmann and J. C. Gyllenhaal and D. M. Gallagher and W. W. Hwu, "Characterizing the Impact of Predicated Execution on Branch Prediction", Proceedings of the 27th International Symposium on Microarchitecture, MICRO-27, December 1994, pp 217-227

[58] C. McNairy, and D. Soltis, "Itanium 2 Processor Microarchitecture", IEEE Micro, March/April 2003, pp.44-55

[59] S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufman, 1997

[60] B. R. Nickerson, "Graph Coloring Register Allocation for Processors with Multi-Register Operands", Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI'90), June 1990, pp. 40-51

[61] C. Norris, and L.L. Pollock, "Register Allocation over the Program Dependence Graph" , Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94) , June 1994, pp. 266-277

[62] J.C.H Park, and M.S. Schlansker, "On predicated execution", Tech. Rep. HPL-91-58, HP Laboratories, Palo Alto, CA, 1991 (May)

[63] T.A.Proebsting and C. N. Fischer, "Probabilistic Register Allocation", Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92) , June 1992, pp 300-310

[64] M. Poletto, and V. Sarkar, "Linear Scan Register Allocation", ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 21, No. 5, 1999, pp. 895-913

[65] B. R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops", Proceedings of the 27th International Symposium on Microarchitecture MICRO-27, Dec. 1994, pp. 63-74

[66] B.R.Rau, M. Lee, P.P.Tirumalai, and M.S. Schlansker, "Register Allocation for Software Pipelined loops", Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92) , June 1992, pp 283-299.

[67] M.S. Schlansker, and B. R. Rau, "EPIC Explicitly Parallel Instruction Computing", Computer, February 2000, pp.37-45

[68] B. Scholz, and E. Eckstein, "Register allocation for irregular architectures", Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems, June 2002, pp. 139-148

[69] A. Settle, D. Connors, G. Hoflehner, and D. Lavery, "Optimization for the Intel Itanium Architecture Register Stack", Proc. of the international symposium on Code generation and optimization, CGO 2003, pp. 115-124

[70] A. Settle, D. Connors, G. Hoflehner, and D. Lavery, “Compiler Controlled Register Stack Management for the Intel Itanium Architecture”, EPIC 3 W orkshop, 2004

[71] J.W. Sias, W -m. W. Hwu, and D. I. August, "Accurate and Efficient Predicate Analysis with Binary Decision Diagrams", Proceedings of the 33rd International Symposium on Microarchitecture MICRO-33, December 2000, pp. 112-123

[72] M.D.Smith, N. Ramsey, and G. Holloway, “A Generalized Algorithm for Graph-Coloring Register Allocation”, Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’04), June 2004, pp. 277-288

[73] P.A. Steenkiste, and J. L. Hennessy, “A Simple Interprocedural Register Allocation Algorithm and Its Effectiveness for LISP”, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 11, No. 1, January 1989, pp. 1-32

[74] N.J. Warter, S.A.Mahlke, W.W. Hwu, and B.R. Rau, “Reverse if-conversion”, Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’93), June 1993, pp. 290-299

[75] R. D. Weldon, S. S. Chang, H. Wang, G. Hoflehner, P. H. Wang, D. M. Lavery, and J. P. Shen “Quantitative Evaluation of the Register Stack Engine and Optimizations for Future Itanium Processors”, Interaction between Compilers and Computer Architectures (Interact), 2002, pp. 57-67

[76] S. Winkel, “Optimal Global Instruction Scheduling for the Itanium® Processor Architecture”, Dissertation, Univ. des Saarlandes, 2004

[77] Y. Wu, L-L. Chen, R. Ju, and J. Fang, “Performance potentials of compiler-directed data speculation.”, 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2003, pp. 22-31

[78] L. Yang, S. Chan, G.R. Gao, R. Ju, G-Y. Lueh, Z. Zhang, ”Inter-Procedural Stacked Register Allocation for Itanium® Like Architecture”,

[79] R. Zahir, J. Ross, D. Morris, and D. Hess, "OS and Compiler Considerations in the Design of the IA-64 Architecture", Proceedings of the 9th International Conference on Architectural Support of Programming Languages and Operating Systems, ASPLOS 2000, pp. 212-221

Index

Curriculum Vitae

Gerolf Fritz Hoflehner
434 Galleria Dr. Unit 04
95134 San Jose, CA

(Home)	+1-408-435-5066
(Cell)	+1-408-506-4240
(Business)	+1-408-765-8380
(Email)	gerolf.f.hoflehner@intel.com

Employment

2009 (Oct.) – , Research Scientist, Intel Corp. (Research Lab)

- Binary translation technology and microarchitecture research

2008 (Dec.) -2009 (Sep.), Software Engineer, Intel Corp. (Binary Translation Team)

- Implemented simple x862x86 translator for enabling BT system
- Defined initial binary translation system validation and testability methodology

2005-2008 (Nov.), Team and Project Lead, Intel Corp. (Compiler Team)

- Led team of 3 engineers on code generation, optimizations and performance measurement/analysis. Projects included register allocation, if-conversion, classical optimizations, EH and feedback-based optimizations
- Managed customer relations and training for Fujitsu/Japan and Oracle/US
- Drove CPU2006 performance analysis (>30% compiler gain on CPU2006), optimizations, publication support
- Designed and implemented predicate-aware and pipeline-aware dataflow analysis, full hazard checking and speculated code analysis algorithm

1999 (Apr.) -2004, Software Engineer, Intel Corp. (Compiler Team)

- Drove Oracle code analysis and compiler performance tuning. Key contributor to >40% compiler gain on TPC-C resulting in world-record setting 4P TPC-C publications on Itanium Linux systems
- Developed optimizations for Itanium register stack, opportunistic interprocedural register allocation, predication oracle, classical optimizations and enabled PQS-based predicate-aware register allocation
- Drove bug dispatching, SPEC and customer code analysis, compile-time task force, local register allocation project, and 4 intern projects

1998-1999, Software Engineer, Siemens Pyramid @Sun Microsystems, Menlo Park

- Ported non-optimizing compiler backend to Itanium on Sun/Solaris
- Planned and oversaw tool deliverables in joint Siemens Pyramid/Sun project
- Developed prototype instruction scheduler for Itanium

1996-1998, Software Engineer, Siemens Pyramid, San Jose, CA

- SPEC95 performance work on pyrC6.0 C/C++ compiler (> 3% gain on integer suite from new and tuned compiler optimizations)
- Oracle performance tuning for cached TPCC setup (Oracle 7.3.2) for R10K. Collected feedback profiling and tuned compiler optimization for 2-3% gain
- Ported MIPS tool chain (UCODE code generator, optimizer, assembler, linker) to 64bit model

1990 (Nov.) -1995, Software Engineer, Siemens Nixdorf Corp., Munich/Germany

- IBM 370 compiler development: Re-engineered big parts of proprietary IBM 370 compiler backend to improve maintainability, added compiler optimizations (sign extensions, set of classical optimizations), enhanced register allocation and instruction selection
- Developed MIPS installation compiler for MIPS RxK
- Supervised students on mathematical library ports (to MIPS) and compiler projects

1988-1990 (Apr.), Intern, Siemens, Munich

- Fast mathematical library development (ULP accurate in double precision for set of elementary mathematical functions) in IBM 370 assembly. Shipped in mathematical library with C/C++/Fortran/Pascal product compilers.
- Database application development for CAD product

1987-1988, Student Program, Siemens, Munich

- Four 2-3 month internships with 2-4 week training classes (mostly in proprietary tools and software packages)

1987-1988, Tutor and teaching assistant at Technical University Munich

- Linear Algebra homework classes for math and computer science students
- Analysis II, III student homework for math students

Paper:

[12] Gerolf Hoflehner: Strategies for Predicate-Aware Register Allocation, Compiler Construction CC 2010, LNCS 6111, pp. 185-204

[11] Darshan Desai, Gerolf Hoflehner, Arun Kejariwal, Daniel M. Lavery, Alexandru Nicolau, Alexander V. Veidenbaum, Cameron McNairy: Performance Characterization of Itanium® 2-Based Montecito Processor. SPEC Benchmark Workshop 2009: 36-56

[10] Arun Kejariwal , Gerolf Hoflehner, Darshan Desai, Daniel M. Lavery, Alexandru Nicolau, Alexander V. Veidenbaum: Comparative characterization of SPEC CPU2000 and CPU2006 on Itanium architecture. SIGMETRICS 2007: 361-362

[9] Peng-fei Chuang, Howard Chen, Gerolf F. Hoflehner, Daniel M. Lavery, and Wei-Chung Hsu: Dynamic Profile Driven Code Version Selection, Interaction between Compilers and Computer Architectures 2007 (Interact 11)

[8] Matthew Bridges, Howard Chen, Gerolf Hoflehner, Daniel Lavery "Fusing Instructions to Reduce Resource Usage in If-Converted Regions", Fifth Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-5), 2006

[7] Gerolf Hoflehner, Knud Kirkegaard, Rod Skinner, Daniel M. Lavery, Yong-Fong Lee, Wei Li: Compiler Optimizations for Transaction Processing Workloads on Itanium® Linux Systems. MICRO 2004: 294-303

[6] Alex Settle, Daniel A. Connors, Gerolf Hoflehner, Daniel M. Lavery: Compiler Controlled Register Stack Management for the Intel Itanium Architecture, Third Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-3), 2004

[5] Alex Settle, Daniel A. Connors, Gerolf Hoflehner, Daniel M. Lavery: Optimization for the Intel® Itanium ®Architecture Register Stack. CGO 2003: 115-124

[4] Gerolf Hoflehner, Daniel M. Lavery, David C. Sehr: The compiler as a validation and evaluation tool. Electr. Notes Theor. Comput. Sci. 82(2): (2003)

[3] R. David Weldon, Steven S. Chang, Hong Wang, Gerolf Hoflehner, Perry H. Wang, Daniel M. Lavery, John Paul Shen: Quantitative Evaluation of the Register Stack Engine and Optimizations for Future Itanium Processors. Interaction between Compilers and Computer Architectures 2002: 57-67

[2] Shih-Wei Liao, Perry H. Wang, Hong Wang, John Paul Shen, Gerolf Hoflehner, Daniel M. Lavery: Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. PLDI 2002: 117-128

[1] Jay Bharadwaj, William Y. Chen, Weihaw Chuang, Gerolf Hoflehner, Kishore N. Menezes, Kalyan Muthukumar, Jim Pierce: The Intel IA-64 Compiler Code Generator. IEEE Micro 20(5): 44-53 (2000)

Conferences:

- Compiler Round Table (Organizer and Host, @Gelato, San Jose, 2005)
- Co-chair Epic Workshop Epic-4 (San Jose, 2005)
- Co-chair Epic Workshop Epic-3 (San Jose, 2004)
- Reviewer for CGO, Micro, Journal of Computing

US Patents:

7,647,482 Method and apparatus for dynamic register scratching, 2010
7,617,495 Resource-aware scheduling for compilers, 2009
7,603,546 System, method and apparatuses for dependency chain processing, 2009
7,398,521 Methods and apparatuses for thread management of multi-threading, 2008
7,328,433 Methods and apparatuses for reducing memory latency in a software application, 2008
7,260,705 Apparatus to implement mesocode, 2007
7,228,528 Building inter-block streams from a dynamic execution trace for a program, 2007
6,907,601 Method and apparatus for inserting more than one allocation instruction within a routine, 2005

Education:

1991 Diplom in Mathematik (Nebenfach Informatik), Dipl. Math. Univ. , Technische Universität München (Master in mathematics with minor in computer science at Technical University Munich/Germany)
Thesis: "Eine Klasse allgemeiner Zetafunktionen und Ihre Funktionalgleichung" (in German)
1986 Vordiplom Mathematik, Dipl. Math. Cand., Technische Universität München
1984 Abitur, Viscardi Gymnasium, Fürstenfeldbruck/Germany

Memberships:

ACM/IEEE/AMS/MAA

Personal Information:

Citizenship:	Austria
US Visa:	Permanent Resident
City of Birth:	Linz/Austria
Date of Birth:	29.08.1965
Interests:	Marathon, Skiing, Hiking, Travel