Strategies for Predicate-Aware Register Allocation

Gerolf F. Hoflehner

Intel Corporation
2200 Mission College Blvd
Santa Clara, CA 95054
gerolf.f.hoflehner@intel.com

Abstract. For predicated code a number of predicate analysis systems have been developed like PHG, PQA or PAS. In optimizing compilers for (fully) predicated architectures like the Itanium® 2 processor, the primary application for such systems is global register allocation. This paper classifies predicated live ranges into four types, develops strategies based on classical dataflow analysis to allocate register candidates for all classes efficiently, and shows that the simplest strategy can achieve the performance potential provided by a PQS-based implementation. The gain achieved in the Intel® production compiler for the CINT2006 integer benchmarks is up to 37.6% and 4.48% in the geomean.

Keywords: Register Allocation, Predication, Compiler, Itanium processor, EPIC, PQS.

1 Introduction

Register allocation solves the decision problem which symbolic register ("candidate") 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 symbolic register. Solutions of 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. Then a coloring is a mapping of a large number of symbolic register candidates to a small, finite set of physical registers. Chaitin describes - in "broad brush strokes" - the fundamental building blocks of coloring allocators [6]. Eminent is the interference graph that encodes the information when two symbolic registers *cannot* be assigned the same physical register. In this case, they are said to *interfere*. A node in this undirected graph is a live range, which represents a candidate and the program points at which the candidate could be allocated a register. A live range is typically modeled by the outcome of two dataflow algorithms: a backward live variable analysis and a forward available variable (or reaching definition) analysis. The live range consists of all program points where the symbolic register is both *live* and *available*. To allocate as many symbolic registers to machine registers as possible the allocator must determine the start of a live range and its interferences precisely. Both problems are harder to solve on predicated architectures like the Itanium processor ("IA-64").

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 185–204, 2010.
© Springer-Verlag Berlin Heidelberg 2010

1.1 Predication

Predication is the conditional execution of an instruction guarded by a qualifying predicate. For example, 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. On IA-64 almost all instructions are predicated. As (almost) fully predicated architecture IA-64 supports if-conversion. If-conversion is a compiler optimization that can eliminate conditional forward branches and their potential branch mis-prediction penalty (Fig. 1). The instructions dependent on the branch are predicated up to a merge point in the original control-flow graph. This eliminates the conditional branch and converts control dependencies (instructions dependent on the branch) into data dependencies (between qualifying instruction predicates) (Allen et al. [2]). As a result if-conversion transforms a control-flow region into a linear ("predicated") code region ("hyperblock"). The paths in the control-flow region become execution traces in the predicated code. In the predicated region all paths of the original region overlap and predicated instructions make it harder for the register allocator to find the start of a live range and determine its precise interferences.

1.2 Overview

The rest of the paper is structured as follows: section 2 gives the background on coloring allocators and the Itanium architecture. Section 3 presents register allocation for predicated code. This section is the core of the paper and presents four classes of predicated live ranges, two methods of precise live tracking interference tracking, and three implementation strategies. Section 4 discusses measurement setup and results. Section 5 reviews related work. Section 6 has conclusions.

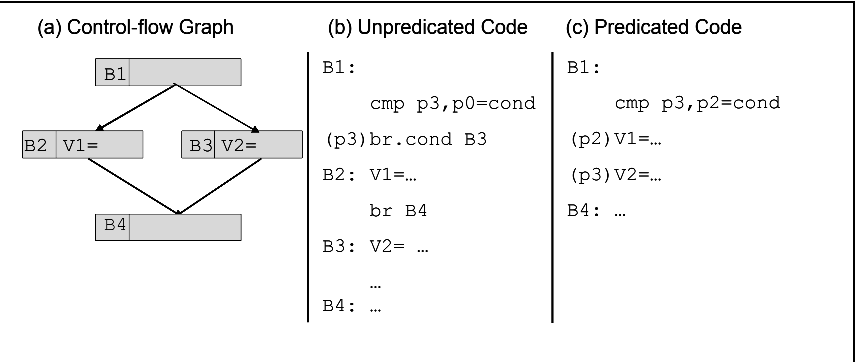

Fig. 1. Example with control-flow graph, non-predicated and predicated code

2 Background

This section gives the background on basic coloring allocator and the Itanium architecture.

2.1 Chaitin-Style Register Allocation

A Chaitin-style graph-coloring algorithm has six phases (Fig. 2): “renumber”, “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 candidates. It also ensures contiguous numbering of candidates which reduces 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 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. This is the case when their live ranges intersect. The number of edges incident with a node is 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, and the vector lists all neighbors of the node.

The next phase, “coalescing” (aka “subsumption”, “node fusion”), is an optimization that is not needed for solving the register allocation problem, but is part of 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

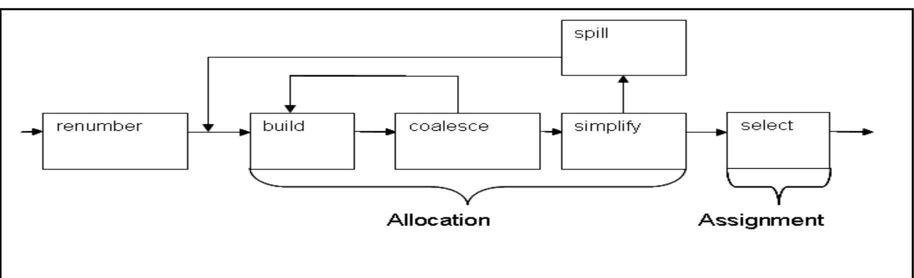

Fig. 2. Chaitin’s model for a coloring allocator

and eliminates the move instruction, since source and destination get assigned the same register. In Chaitin's original implementation any possible pair of nodes is coalesced. 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 filter candidates that can be certainly allocated to machine registers: when a register candidate has fewer than K interference edges (a *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 simplification is in a blocking state. It transitions out of a blocking state using a heuristic based cost 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. 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.

2.2 Itanium® 2 Processor Architecture

The Itanium processor family 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 processor. Instructions are grouped in fixed-size bundles. These are simple structures that contain three instructions and information about the functional unit each instruction must be issued. IA-64 is a 64bit computer architecture distinguished by a fully predicated instruction set, 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. Predicates enable the compiler to remove branches ("branch barrier removal"), control speculation allows it to hoist load instruction across branches ("breaking the branch barrier"), and data speculation makes it possible to hoist load instruction 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 register usage. In general exploiting instruction-level parallelism using Itanium features increases register pressure and poses new challenges for the register allocator. To support the EPIC paradigm the Itanium processor provides a large number of architected registers. 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. The floating point and predicate register files contain rotating registers f32-f127 and p16-p63 respectively. Unique for Itanium are the 96 stacked integer

registers, r32-r127, which are controlled by a special processor unit, the Register Stack Engine (RSE).

3 Register Allocation for Predicated Code

The IA-64 architecture is a fully predicated architecture with 64 predicate registers [10]. Each instruction (with some exceptions like the alloc instruction) is guarded by a qualifying predicate. A fully predicated architecture supports if-conversion, an optimization that can eliminate 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. The proper representation of a predicated region is a hyperblock, which is a predicated single entry multiple exit region. The compiler picks a single entry acyclic control-flow region as a candidate region for if-conversion. It may have multiple exits. Basic blocks where the paths originating from the single entry meet, are merge points and mark a potential end node for the region former. Typically the compiler has a threshold for the number of blocks in a region. The decision whether or not to if-convert a candidate region is driven by a predication oracle. It computes 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. The register allocator must handle predicated code formed from control-flow graph regions. This section 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. The allocator based on "use-and-partition tracking" is equivalent to a PQS-based allocator, but simpler allocators are investigated as well.

3.1 Impact of Predicated Code

On a predicated architecture completeness and soundness of live variable and disjoint live range information are harder problems than for non-predicated 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 non-predicated code a definition is the start of a live range. This is not necessarily true for predicated code. In Fig. 3 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.

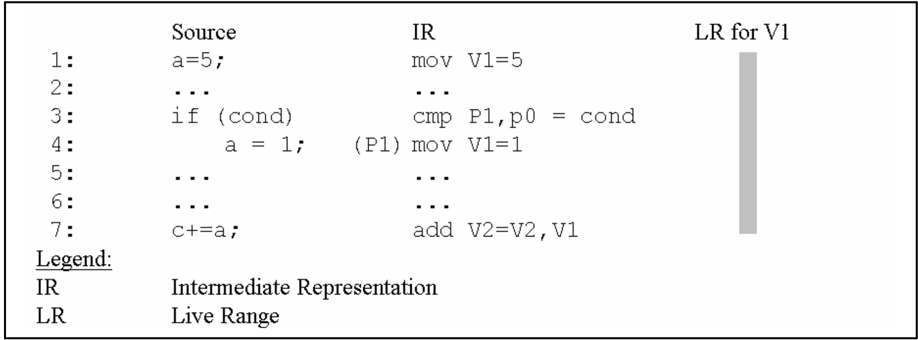

Fig. 3. Example with source code, predicated code and predicated live range. The bar at the right represents the actual live range of V1.

On the other hand, when 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 non-predicated 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 must be so since the variable could be 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. In acyclic code, live range extension cannot occur because a live range cannot extend to a program point where it is not available. Fig. 4 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 next allocation round 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 a potential run-time performance loss due to extra register pressure, but also a stability reason why a predicate-aware allocator must recognize the start of

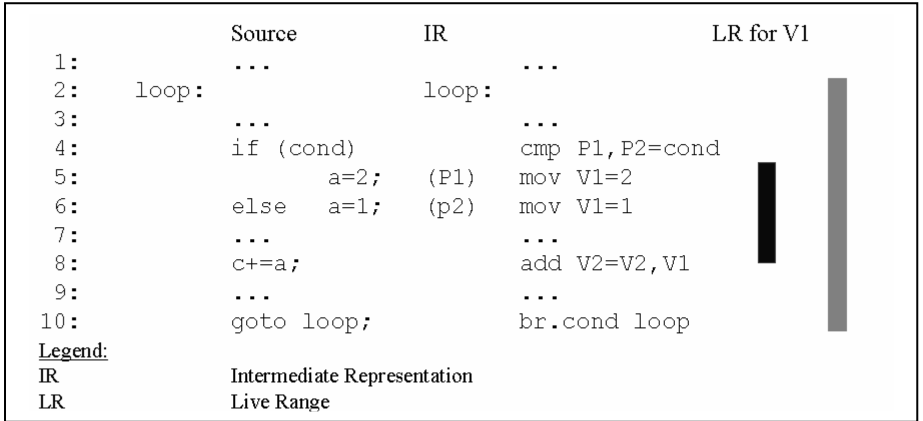

Fig. 4. Example for live range extension in predicated code. Short bar represents exact live range, while the long bar represents the extended live range of V1.

predicated live ranges. This impacts live range analysis (in particular, live variable analysis) and interference graph construction, which happens in the "build" phase of the allocator.

In non-predicated 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 Predicate Data Base (PDB) for disjointness information when it constructs the interference graph: if the sets of predicates that guard two live ranges L1 and L2 are disjoint, no interference edge needs to be added between them.

Interference graph construction for predicated code is similar to non-predicated code: it is a backward scan of the instructions in a basic block with a single, non-predicated live vector initialized with the live-at-exit candidates. For each candidate the guarding predicates (=qualifying predicates of the instruction that references the candidate) are recorded as predicate sets associated with the live range in a separate table. The compiler routine checking for interference takes the qualifying predicate, 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. Interference is recorded accordingly in the interference graph. A defined variable is removed from the live vector when it is recognized as the start of its 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. This is the set of predicates under which a candidate is live and is kept in the table mentioned above. The live vector representation in the predicate-aware allocator does not (need to) change and can remain predicate-*unaware*.

Finally, a predicate-aware live variable analysis *must* treat predicated live ranges conservatively across back edges. For example, in Fig. 5 variable B would be 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 is live under P1, the allocator would recognize A and B as disjoint and could assign them the same physical register. In this case

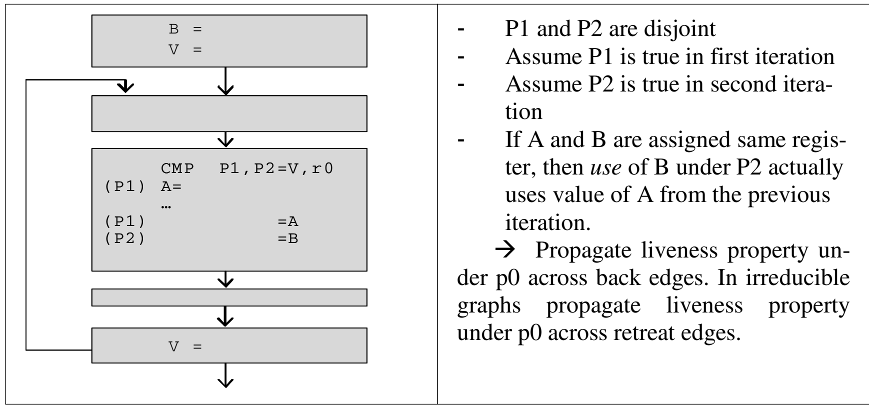

Fig. 5. Propagating liveness property under p0 on back edge

the assignment to A in the first iteration would overwrite B, which is used in the second iteration. When the live range of B becomes live under p0, which is the True predicate, then – since p0 interferes with every predicate – the live ranges for A and B are no longer disjoint.

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. Removing retreat edges gives an acyclic graph, but global disjointness is not necessarily consistent with local disjointness in an arbitrary acyclic region of the original graph.

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

In predicated code live variable analysis and interference graph construction must reason about predicates. Both must find the start of a live range. Interference calculation must also recognize disjoint live ranges. The predicate query system (PQS) as described in Gillies et al. [8] 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. The partitions represent execution paths and are derived by a single traversal of the control flow graph. For each block in the graph that has two or more successors or predecessors the partition $P = P1 \mid P2 \mid \ldots \mid PN$ 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 (predecessors).

(a) Source Code Fragment

```
 1: S1: D=…
 2:    if (x<y) {
 3:    S2: a=…
 4:        C=…
 5:        if (a==3) {
 6:        S4: A=…
 7:            B=…; B1=…;
 8:            …=C
 9:        } else {
10:        S5: A=…
11:            B=…; B1=…;
12:            …=C
13:        }
14:        …=B;…=B1;
15:    } else {
16:    S3: A=…
17:        b=…
18:        if (b>10) {
19:        S6: …
20:        }
21:    }
22:    …=D;…=B; …=A;
23:    if (i!=5) {
24:        S7:…
25:    } else {
26:        S8:…
27:    }
```

(b) Predicated Code Fragment

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

Fig. 6. Example to illustrate liveness under predicate sets. C source code and corresponding predicated code in an intermediate representation.

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 is a single 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 $B1 \rightarrow B2$ is critical. The inserted 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 (Park and Schlansker [16]). The characteristic of the RK algorithm is that it assigns the same block predicate to a set of control-equivalent basic blocks. Two basic blocks B1 and B2 are control-equivalent if 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 predicate relations derived from the control-flow graph in the PPG.

We use a more elaborate version of the example in Johnson and Schlansker 12] to illustrate the predicate partition graph and PQS. Fig. 6 shows source code and predicated code of our example, while Fig. 7 has the control-flow graph including block predicates and PPG. Control-equivalent basic blocks are assigned the same block predicates. The edge from Block 3 to Block 8 is critical, and the completion phase inserted JS-Block B6' on the edge. The acyclic predicate partition graph corresponding to the control-flow graph fragment is shown in Fig. 7. Partitions $P1 = P2 \mid P3$ (edges "a"), $P1 = P7 \mid P8$ (edges "e") and $P3 = P6' \mid P6$ (edges "c") are

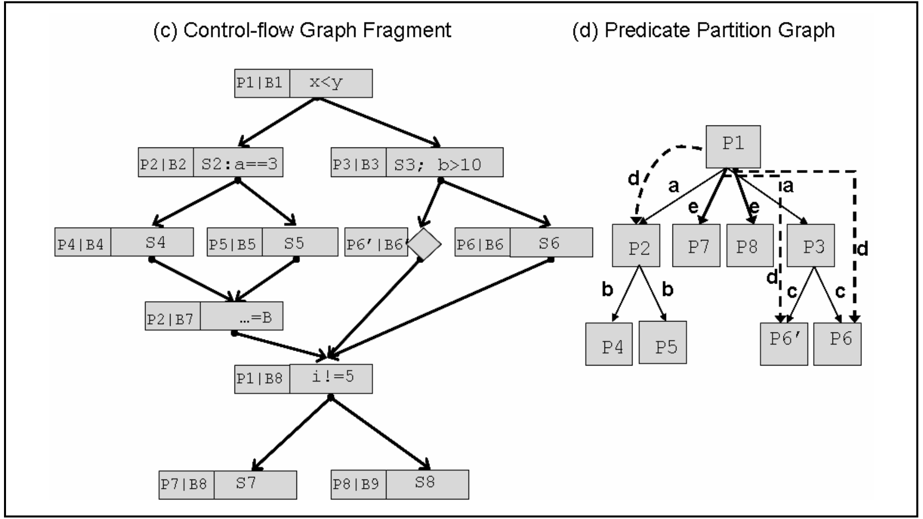

Fig. 7. Control-Flow Graph and PPG for the example in Fig. 6

forward partitions, partition $P1 = P2 \mid P6' \mid P6$ (edges "d") is a backward partition and $P2 = P4 \mid P5$ (edges "b") is both, a forward and a backward partition.

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. Fig. 10 shows two predicated live ranges A and B and their predicated sets during a PQS-based backward traversal of instructions 1-20 in the predicated code fragment of the example. The interference graph construction uses this backward traversal to find and record live range interferences.

PQS is powerful, but it comes at a cost. First, it requires the construction of the predicate partition graph, which is linear in space and time of the number of basic blocks. Second, unlike classical live variable analysis, which operates on basic blocks, the granularity for PQS-based predicated live variable dataflow is an instruction, so customized dataflow routines are required. Finally, the PQS queries are invoked at every predicated instruction. These cost factors motivate alternative solutions.

3.3 A Family of Predicate-Aware Register Allocators

This section assumes all predicated code is compiler generated. We propose a family of predicate-aware allocation schemes based solely on classical register allocation 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 the qualifying predicates for the definitions and uses of a live range match or a definition dominates all uses. In other cases, partitions can be pre-computed at each use on demand. Building and repeatedly querying the PPG is not necessary. In particular we will identify four types of predicated live ranges – match, dominate,

partition, overlap - two methods for interference modeling – “simple” and “complex”, and three implementation strategies. In Strategy 1, only match and dominate live ranges are modeled precisely. Strategy 2 models all simple live ranges precisely, and Strategy 3 models all live range precisely, which is equivalent to a PQS-based implementation. The four types of predicated live ranges are recognized in the original control-flow region.

There are four fundamental relations between predicated definitions and uses (Fig. 8). 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 Fig. 8, 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” I n Fig. 8).

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 is used to form a partition.

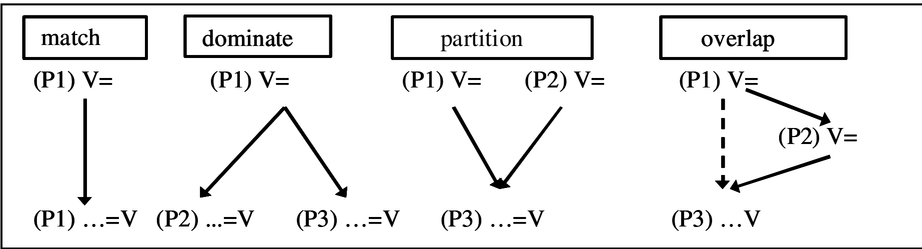

Fig. 8. Classification of predicated live ranges

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

Fig. 9. Example from Fig. 6 and two variables A and B and their predicate sets as seen by interference graph construction for a) partition based tracking and b) PQS-based tracking

Our example Fig. 6 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, the first definition of a variable (= start of its live range) can be marked in a forward sweep over all instructions in the (linear) if-converted region, using the live-at-entry vector and recording definitions: at a given predicated instruction if the variable defined is not live-at-entry and no other definition of the variable has been seen, the first definition of the variable has been found.

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

Strategy 1: Dominate-or-Match
The qualifying predicates of instructions that use a variable form the predicate set of that variable. 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.

Fig. 9 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.

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

Theorem 1. (Characterization of Simple Live Ranges)

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 each partition predicate.
- 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 these cases 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 disjointness in the predicated region is precise with respect to instructions (variables) 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.

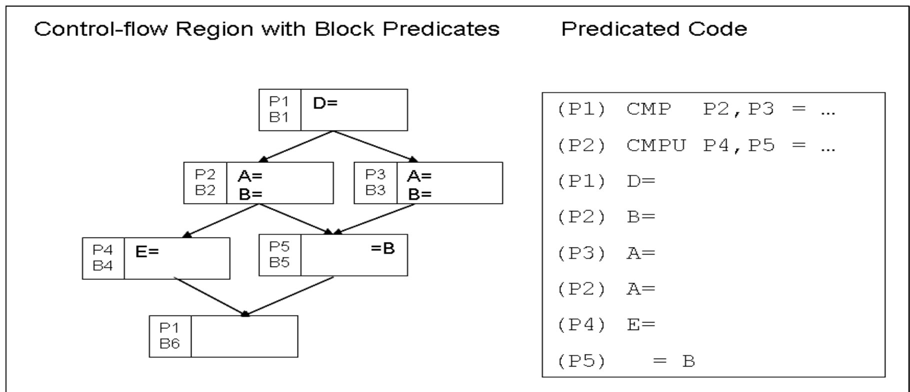

Fig. 10. Tracking live variable of B under partition P2|P3 would result in extra interference with E in the predicated code. Tracking B under P5 does not cause this interference since P4 and P5 are disjoint.

Therefore tracking the live range under partition predicates cannot introduce extra interferences. The case of two definitions that don't from a partition and reach a use can be reduced to the partition case. Since there is only one path that contains both definitions, there must exist a basic block with a predicate disjoint to the "second" definition predicate (= block predicate of block containing the "second" definition). Since the region is assumed to be complete (=JS blocks inserted as needed), the block can be chosen so that the block containing the "first" definition predicate dominates it. Inserting a pseudo definition in the selected block ensures the partition property: since the qualifying predicate of the pseudo definition 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 can be reduced to this special case.

Consider the example code in Fig. 10 to visualize an imprecise interference when the use does not post-dominate definitions. 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.

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.

In case definitions 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.

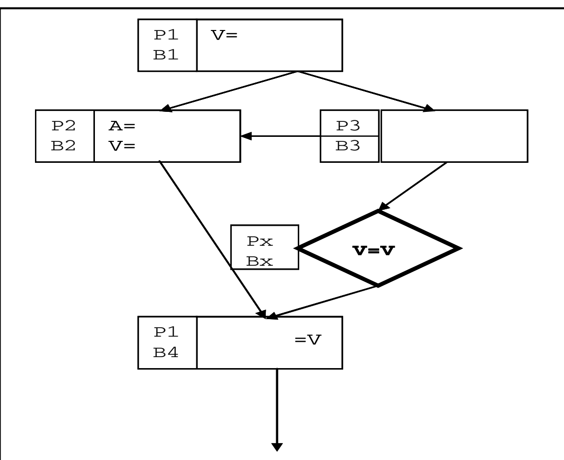

Fig. 11. Complex live range tracking and splitting for live range of variable V. Only the control-flow graph of the region is shown before if-conversion. An identical move for variable V in basic block Bx with block predicate Px splits the original complex live range for V into a partition and a dominate live range.

Fig. 11 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 $1 \rightarrow 2$ and $1 \rightarrow 3 \rightarrow 2$. The disjoint partition for the definition in block B4 is P2| Px, where Px is a JS block. Since P1 does neither dominate nor match Px, tracking under Px would not find the start of the live range. The trick is adding 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. At the definition in block Bx the predicate set for V contains Px together with the associated partition P1|P1', which corresponds to definitions of V reaching the use. The original live range has been split into a "partition" live range (see Fig. 8), which is covered by Strategy 2, and a "dominate" live range.

From the discussion is it clear that Strategy 3 models interference precisely when a live range has two definitions on more than one execution path. The identity move can be inserted where the two definitions merge. Note that translating out of an SSA representation (15]) before if-conversion would yield the moves. The general case is

Theorem 2. (Characterization of Complex Live Ranges)

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

- Use predicate does not post-dominate partition predicates (1)
- When definitions overlap on more than one path, the live range can be split and handled by Strategy 2 or the case above. (2)

Proof

Preciseness for first case (1) 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 the use. Partition tracking asserts that there is no conservative (false) interference with variables in instructions on the path from the entry code to the definition. Interference calculation 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 (actually) disjoint from the live candidate.

Overlapping live ranges on multiple paths can be split into to simpler live ranges: Assume the live range has $n > 2$ definitions. Like in Fig. 11 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 completes the proof of the theorem since it holds in the case n=2. □

We identified four types of predicated live ranges – match, dominate, partition and overlap – and two methods for interference modeling – "simple" and "complex", and three implementation strategies. The simple method covers all "match" and "dominate", and some partition and overlap live ranges ("simple live ranges"), while the complex method handles remaining partition and overlap live ranges ("complex live ranges"). Strategy 1 models only "match" and "dominate" live ranges precisely. Strategy 2 models all simple live ranges precisely, and finally Strategy 3 models all live range precisely. Strategy 3 is equivalent to a PQS-based implementation.

4 Results

We obtained the performance data on a 1.6 GHz Montecito processor using the Intel Fortran/C++ optimizing compiler (version 11.1). The detailed configuration is listed in Table 1. The benchmark suite is CINT2006, a popular industry-standardized CPU-intense suite used by OEMs for stressing a system's processor, memory subsystem and compiler. We only show CINT2006 data, since the CFP2006 data are similar.

Table 1. 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)

The gains from predicate-aware register allocation are for two different implementations. In section 3.3 we classified four types of predicated 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 interference modeling. The basic implementation that tracks liveness under the qualifying use predicates and marks first predicate definitions (Strategy 1) gives practically identical run-time performance as the PQS-based implementation. The difference ("Delta") between the methods shown in Table 2 is within the run-to-run variation of the benchmarks.

Table 2. Performance Gains from Predicate-Aware Allocation on CINT2006

Benchmark	Basic allocation	PQS allocation	Delta
400.perlbench	37.67%	37.67%	0.00%
401.bzip2	5.01%	5.01%	0.00%
403.gcc	1.71%	1.45%	-0.26%
429.mcf	0.93%	0.93%	0.00%
445.gobmk	2.91%	2.91%	0.00%
456.hmmer	1.20%	1.20%	0.00%
458.sjeng	8.01%	8.01%	0.00%
462.libquantum	0.00%	0.37%	0.37%
464.h264ref	0.00%	1.03%	1.03%
471.omnetpp	0.40%	0.40%	0.00%
473.astar	0.97%	0.00%	-0.96%
483.xalancbmk	0.00%	0.00%	0.00%
Geomean	4.48%	4.50%	0.01%

The outlier in Table 2 is the 37.67% gain in 400.perlbench. This is due to reduced RSE traffic in S_regmatch, the hottest (and self-recursive) function of the benchmark. Without a predicate-aware allocator all 96 register on the register stack get allocated. With a predicate-aware allocator only about half the number of registers is used. The increase in register pressure without the predicate-aware allocator is explained with live range extensions in loops (see Fig. 4).

The performance data suggest that basic predicate-awareness in the coloring allocator reaps the performance benefits. The performance gain from the simplest predicate-aware allocator (Strategy 1) and PQS-based predicate-aware allocator match. The basic predicate-aware allocator models precise interference only for match and dominate live ranges, but is conservative for all partition and overlap live ranges. For the experiment, live ranges were first 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 within the region. When a live range falls into multiple classes, only the "most complex" class (overlap > partition > dominate > match) gets accounted for. The data for predicated live ranges distribution is in Table 3. 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%. The data in the tables were collected for all predicated live ranges in all predicated regions of a benchmark. Since only relatively few partition and overlap live ranges exist, a system like PQS or complex live range tracking is not necessary for precise interference modeling. The data and code analysis suggests that the complex tracking cases are very rare (less than 2.5% for all benchmarks). If conservative disjointness for partition and overlap live ranges is a concern, Strategy 2 models more than 97.5% of the predicated live ranges precisely.

Table 3. Distribution of Predicated Live Ranges

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%

5 Related Work

There are a number of approaches to represent predicate relations in a compiler. The IMPACT compiler uses the Predicate Hierarchy Graph (PHG) (Mahlke et. al. [13]). For each definition of a predicate the PHG tracks the predicates that guard the definition. It can also handle OR-expressions and is applied to code in hyperblock regions. A hyperblock is a predicated superblock, which is an acyclic single entry multiple exit region. A more sophisticated approach than the PHG is the predicate query system (PQS) (Gillies et al. [8], Johnson and Schlansker [12]). It uses the predicate partition graph to determine predicate relations. PQS can determine accurately predicate relations that can be expressed as logical partitions. Here 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. Both PHG and PQS use approximations in the analysis of already predicated code. In this case the code is not derived from the control-flow graph, but instead supplied by the user (in the case of assembly code) or an earlier compiler phase. When the code is derived from the control-flow graph, PQS can represent predicate disjointness information accurately in acyclic regions. More subtle predicate analysis methods that derive accurate predicate relations for already predicated code have been developed also. Eichenberger [7] represents logical predicate relations, so called P-facts, and determines predicate relations in a logic solver. He applies this information for register allocation in hyperblocks. Sias et al. [18] developed the predicate analysis system (PAS), which is as accurate as Eichenberger, but can determine predicate relations globally using a BDD solver. In contrast to the approaches described in literature, this paper makes no attempt to address the general predicate relation problem. It makes the assumption that predicated code is derived from acyclic control-flow graph regions. This holds in general for compilers. Our paper shows that classical interference calculation can be extended in various degrees of accuracy to model interference of predicated live ranges precisely. PQS-based allocation is used for reference.

6 Conclusions

This paper classified predicated live ranges into four types: match, dominate, partition and overlap. It described implementation strategies based on classical dataflow analysis to allocate register candidates for all classes efficiently and precisely. We implemented a basic predicate-aware allocator that models match and dominate live ranges precisely, and partition and overlap live ranges conservatively. We compared the basic allocator to a PQS-based allocator and found practically no performance difference on CINT2006. In this case investing in a sophisticated predicate database and query system for a predicate-aware allocator is not necessary. The reason is that only a small portion of live ranges in predicated code require complex live range tracking for precise interference modeling. The gain achieved from predicate-aware register allocation on the CINT2006 integer benchmarks is up to 37.6% and 4.48% in the geomean for the SPEC base options of the Intel Itanium production compiler.

References

1. Aho, V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools, 2nd edn. Addison Wesley, Reading (2007)
2. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.D.: Conversion of Control Dependence to Data Dependence. In: Proceedings of the 10th ACM Symposium on Principle of Programming Languages, POPL 1983, January 1983, pp. 177–189 (1983)
3. Bharadwaj, J., Chen, W.J., Chuang, W., Hoflehner, G., Menezes, K., Muthukumar, K., Pierce, J.: The Intel IA-64 Compiler Code Generator. IEEE Micro, 44–52 (September/October 2000)
4. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to Graph Coloring Register Allocation. ACM Transactions on Programming Languages and Systems 16(3), 428–455 (1994)
5. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.: Register allocation via coloring. Comp. Lang. 6(1), 47–57 (1981)
6. Chaitin, G.J.: Register Allocation and Spilling via Graph Coloring. In: Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, pp. 98–105 (1982)
7. Eichenberger, A., Davidson, E.S.: Register allocation for predicated code. In: Proceedings of the 28th Annual International Symposium on Microarchitecture, MICRO-28, December 1995, pp. 180–191 (1995)
8. Gillies, D.M., Ju, R.D.-C., Johnson, R., Schlansker, M.S.: Global Predicate Analysis and its Application to Register Allocation. In: Proceedings of the 29th International Symposium on Microarchitecture, MICRO-29, December 1996, pp. 114–125 (1996)
9. Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H., Zahir, R.: Introducing the IA-64 Architecture. IEEE Micro, 12–22 (September/October 2000)
10. Intel Corporation, Intel® Itanium® Architecture Software Developer's Manual, Revision 2.2, vol. 1-3 (January 2006), `http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm`
11. Intel Corporation, Intel® Itanium® 2 Processor Reference Manual (May 2004), `http://download.intel.com/design/Itanium2/manuals/25111003.pdf`
12. Johnson, R., Schlansker, M.S.: Analysis techniques for predicated code. In: Proceedings of the 29th International Symposium on Microarchitecture, MICRO-29, December 1996, pp. 100–113 (1996)
13. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: Proceeding of the 25th Annual International Symposium on Microarchitecture MICRO-25, December 1992, pp. 45–54 (1992)
14. McNairy, C., Soltis, D.: Itanium 2 Processor Microarchitecture. IEEE Micro, 44–55 (March/April 2003)
15. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
16. Park, J.C.H., Schlansker, M.S.: On predicated execution. Tech. Rep. HPL-91-58, HP Laboratories, Palo Alto, CA (May 1991)
17. Schlansker, M.S., Rau, B.R.: EPIC: Explicitly Parallel Instruction Computing. Computer, 37–45 (February 2000)
18. Sias, J.S., Hwu, W.-M.W., August, D.I.: Accurate and Efficient Predicate Analysis with Binary Decision Diagrams. In: Proceedings of the 33rd International Symposium on Microarchitecture MICRO-33, December 2000, pp. 112–123 (2000)