Prematerialization: Reducing Register Pressure for Free

Ivan D. Baev Richard E. Hank David H. Gross

Java, Compilers, and Tools Laboratory
Hewlett-Packard Company
11000 Wolfe Road, Cupertino, CA 95014
{ivan.baev, rick.hank, david.gross}@hp.com

ABSTRACT

Modern compiler transformations that eliminate redundant computations or reorder instructions, such as partial redundancy elimination and instruction scheduling, are very effective in improving application performance but tend to create longer and potentially more complex live ranges. Typically the task of dealing with the increased register pressure is left to the register allocator. To avoid introduction of spill code which can reduce or completely eliminate the benefit of earlier optimizations, researchers have developed techniques such as live range splitting and rematerialization.

This paper describes prematerialization (PM), a novel method for reducing register pressure for VLIW architectures with nop instructions. PM and rematerialization both select "never killed" live ranges and break them up by introducing one or more definitions close to the uses. However, while rematerialization is applied to live ranges selected for spilling during register allocation, PM relies on the availability of nop instructions and occurs prior to register allocation. PM simplifies register allocation by creating live ranges that are easier to color and less likely to spill. We have implemented prematerialization in HP-UX production compilers for the Intel® Itanium® architecture. Performance evaluation indicates that the proposed technique is effective in reducing register pressure inherent in highly optimized code.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *Compilers, Code generation, Optimization*

General Terms

Algorithms, Design, Performance

Keywords

Register allocation, Rematerialization, Register pressure, VLIW, Itanium

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PACT'06, September 16–20, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-264-X/06/0009…$5.00.

1. INTRODUCTION

The much lower access time of physical registers compared to that of memory has always made them a critical processor resource. Although the number of registers has been steadily increasing in each new generation of processors, the register requirements of newly developed and more aggressive compiler optimizations have continued to stress this resource. High-level code transformations such as procedure inlining and loop transformations tend to enlarge the size of regions subject to later optimizations. Operating on larger regions is advantageous for global redundancy elimination and instruction scheduling algorithms because there are more opportunities to remove redundant computations and reorder instructions, respectively. The common side-effect is a tendency towards fewer but larger and more complex live ranges which results in higher register pressure. For example, an implementation of partial redundancy elimination (PRE) is reported to cause about a 25% average increase in register pressure [7].

If the number of overlapping live ranges exceeds the number of available physical registers, selected live ranges must be spilled to and reloaded from memory, which can reduce overall performance. On the other hand, conservatively scaling down the aggressiveness of certain optimizations in order to limit register pressure may lead to missed opportunities and underutilized register resources, resulting in suboptimal performance. Finding the right balance between limiting and tolerating register pressure is thus one of the ever-lasting issues for compiler designers. At a high level, there are two approaches for solving the register pressure problem. We can make the optimizations that precede register allocation more cognizant of register pressure, or we can provide the register allocator with more sophisticated mechanisms for tolerating register pressure.

In the first category, most studies have concentrated on PRE and scheduling. Gupta and Bodik propose a register pressure sensitive redundancy elimination method, which first sets upper limits on allowed register pressure and then performs redundancy elimination within these limits [7]. Choosing the appropriate values for constraints is difficult because different benchmarks require different limits for the best performance. Instead of heuristic limits for guiding redundancy elimination, Zhao et al. employ models of the PRE algorithm, code stream, and register allocator to evaluate the benefit of applying a particular transformation [19]. Simultaneously considering the effect on register pressure and code is shown to be more profitable than paying attention to the register pressure alone; however, the models could become inaccurate if PRE and register allocation are separated by other major code transformations.

Register pressure aware schedulers take into account register requirements during each scheduling step. For example, the unified resource allocator (URSA) method augments the scheduling dependency graph with sequential edges for regions with high register pressure [1]. Touati refines the URSA method by developing better heuristics for computing the maximal register requirement for a schedule [18]. With the goal of reducing register spills, Govindarajan et al. address the problem of generating an instruction sequence with a minimum number of registers, and present an efficient heuristic solution based on chains of overlapping live ranges [6].

The other approach to cope with register pressure is to enhance the register allocator itself. Researchers and compiler practitioners have commonly experimented with various strategies for register rematerialization, live range splitting [2], and region-based register allocation [14][9]. Rematerialization refers to breaking up live ranges of values that are always available, or “never killed”, by introducing one or more definitions close to the uses. Chaitin notes that such values should be recalculated instead of being spilled to and reloaded from memory [5]. Briggs et al. enhance the rematerialization idea by applying it to complex, multi-valued live ranges [3]. In both papers, rematerialization happens within the spill part of the graph coloring register allocation loop; that is, a live range is rematerialized only when it would otherwise be spilled. In contrast, Simpson proposes a register rematerialization pass before the register allocator when a register pressure estimate is considered high [16]. However, it is difficult to correctly approximate register pressure without building an interference graph and attempting to color it. Furthermore, the actual register pressure can change because of coalescing by the register allocator. Any overestimation of register pressure leads to extra rematerialization; similarly, an underestimate may lead to unnecessary spill code.

In this paper we introduce ***prematerialization*** (PM), a novel method for reducing register pressure for VLIW architectures with nop instructions. PM attempts to redefine virtual registers in available nop slots closely preceding the register uses. PM happens prior to register allocation and bears similarities to Briggs-style spill-time rematerialization [3]. The goal of both approaches is to redefine certain “never killed” values close to their uses, but while rematerialization always inserts new definition instructions, which may potentially increase the path length of the final code (both because of the added instructions themselves and because it may not be possible to place them in such a way as to avoid stalls), PM is an opportunistic technique relying on the availability of nop slots in previously scheduled code, and never increases path length. PM simplifies the work of the register allocator by creating live ranges that are easier to color and less likely to spill. PM assists the other techniques for reducing register pressure. For example, prematerialized live ranges are still eligible for rematerialization at spill time, although our experiments show that PM greatly reduces the need for it.

The rest of the paper is organized as follows. The next section introduces features of the Itanium® architecture that motivated our work. Section 3 presents the prematerialization method. A performance evaluation is given in Section 4. In Section 5 we conclude with a summary and a discussion of issues with and possible enhancements to prematerialization.

Table 1. RSE contribution to execution time (in %)

SPECint2000	% RSE	SPECfp2000	% RSE
164.gzip	*0.1*	168.wupwise	*0.1*
175.vpr	*0.2*	171.swim, 172.mgrid	*0.0*
176.gcc	*4.1*	173.applu	*0.0*
181.mcf	*0.1*	177.mesa	*8.0*
186.crafty	*8.2*	178.galgel	*0.1*
197.parser	*2.4*	179.art	*0.0*
252.eon	*2.9*	183.equake	*0.1*
253.perlbmk	*3*	187.facerec	*0.1*
254.gap	*2.6*	188.ammp, 189.lucas	*0.0*
255.vortex	*5.7*	191.fma3d	*0.4*
256.bzip2	*1.1*	200.sixtrack	*0.2*
300.twolf	*0.2*	301.apsi	*1.1*

2. BACKGROUND AND MOTIVATION

The Intel® Itanium® architecture [11] is a VLIW/EPIC architecture that provides a number of features designed for exposing and exploiting instruction-level parallelism (ILP), such as control speculation, data speculation, and predication. A natural side-effect of utilizing these capabilities in addition to aggressive scalar optimization is an increase in register pressure. There are several aspects of the Itanium® architecture that present new challenges and opportunities for effectively managing register pressure. In this section we discuss the two aspects that motivated prematerialization: register files and instruction bundling.

2.1 Register Files

The Itanium® architecture provides a large number of registers to accommodate the needs of applications compiled with aggressive scalar and ILP exposing transformations.

The integer register file contains 128 registers; the lower 32 registers are static and the upper 96 registers are stacked. A procedure may allocate a variable sized register stack frame composed of up to 96 registers. The stack frame is divided into incoming argument registers, local registers, and outgoing argument registers. A subset of the stack frame can be specified to rotate in the context of a software pipelined loop. The registers in the register stack are managed by the register stack engine (RSE). The incoming argument and local registers are seen by the register allocator as being preserved by procedure calls.

The floating-point register file also contains 128 registers. Unlike the integer register file, the upper 96 floating-point registers are not stacked, and all of these 96 registers rotate in the context of a software pipelined loop. The Itanium® calling convention [13] specifies that the upper 96 floating-point registers are scratch, i.e., not preserved across procedure calls.

In an Itanium® processor, high register pressure manifests itself in two ways. The first manifestation consists of *explicit* spills to and fills from memory, as with many other instruction set architectures. The second manifestation consists of execution stalls due to *implicit* spills and fills generated by the RSE. The RSE operates on a circular array of physical registers. When a register stack frame is allocated for a procedure and the number of available registers in the array is less than the size of the frame, execution stalls while the RSE spills registers to memory to make room for the new register stack frame. Similarly, when a

procedure call returns it may be necessary for the RSE to restore the caller's register stack frame from memory. Table 1 shows the percentage of execution time spent in these RSE related stalls for the SPEC2000 integer and floating-point suites, respectively. For a number of these applications the RSE stalls account for a non-trivial component of runtime — as high as 8% in the case of 186.crafty and 177.mesa. The challenge presented by the RSE contribution to runtime has motivated investigations into active management of the register stack frame [15][10].

The partitioning of the floating-point register file presents an interesting challenge to code generation and register allocation. The fact that the upper 96 floating-point registers are scratch and rotating has severe implications to a floating-point live range that spans either a procedure call or a software pipelined loop; such a live range cannot be assigned to any of the upper 96 floating-point registers. Compare this situation with an integer live range that spans either a call or a software pipelined loop: the incoming argument and local registers in the register stack frame are preserved across a call, and not all of the register stack frame necessarily rotates. The additional register allocation freedom for integer registers opens up the opportunity to trade floating-point register pressure for integer register pressure. The live ranges that are particularly well suited to this tradeoff are those whose defining operation is a move from the floating-point register file to the integer register file. On Itanium® this is achieved via a *setf* instruction. Prematerialization of *setf* instructions will be discussed in more detail in Section 3.2.

2.2 Instruction Bundles

The very long instruction word in the Itanium® architecture is a bundle of three slots. Each instruction occupies one slot (usually) or two slots (rarely) in a bundle. The bundle also contains a template that specifies the functional unit type for its instructions. Each instruction is assigned one of five basic instruction types: *M*, *I*, *F*, *L,* or *B*. Twelve distinct templates place limitations on the allowable combinations of instructions within a bundle. For example, the *MIB* template indicates that a bundle contains *M*, *I*, and *B* instructions, in that order. In addition to functional unit types, the template specifies zero, one, or two stop bits within and/or at the end of the bundle. A stop bit indicates to the hardware that instructions on either side of the stop bit are to be executed in different cycles. Instructions between stop bits may be executed in parallel.

Typically the instruction stream is packed into bundles during the instruction scheduling phase of the compiler. When the scheduler is unable to completely fill a bundle with useful instructions the empty slots are filled with nops. Not only do these nops indicate that there exist underutilized functional units on the processor within a given issue cycle, but the template indicates exactly which functional units are available. The existence of nop instructions provides the means by which a live range can be simplified with no impact on path length.

Prematerialization relocates and clones the definition of a "never killed" live range as necessary to provide dominating definitions closer to the uses of that live range. Provided that compatible nops can be found far enough away, in terms of cycles, from each use to meet the latency of the defining instruction, placing prematerialized definitions in nop slots is free: we achieve a reduction in register pressure with no increase in path length.

3. THE PREMATERIALIZATION METHOD

This section describes the algorithmic details of prematerialization. We first introduce the core structure of the method by outlining its major steps and working through an example. We then show how prematerialization is implemented in our HP-UX Itanium® compilers.

3.1 Core Algorithm

The prematerialization phase takes place between the instruction scheduler and the register allocator. We assume that the scheduler has bundled the instructions into explicit long words or templates such that the nops are explicit in the code stream. Prematerialization operates on virtual register live ranges, each having a single definition and any number of uses. The goal is to simplify each suitable live range by inserting one or more copies of the definition into available nop slots preceding the uses and as close as possible to the uses subject to instruction latency. The resulting live ranges are more numerous and less complex, and they occur in an instruction stream that has fewer nop instructions and more computation instructions.

The prematerialization algorithm consists of three steps: candidate selection, candidate prioritization, and definition materialization.

3.1.1 Candidate Selection

Prematerialization begins by collecting the set of candidate live ranges. For each virtual register live range within the procedure, we examine the number and types of its defining instructions and the locations of its uses. Each candidate is required to have a single definition; a virtual register having multiple definitions will not participate in prematerialization. A candidate must have at least one use outside the basic block containing its definition; because of the typically small number of instructions within a basic block, we assume it is not worthwhile to bring a definition closer to its uses in the same basic block.

Candidates eligible for prematerialization are conceptually divided into classes, such that all candidates in a class are "similar" for analysis purposes — e.g., we might require that they define the same kind of register (integer or floating-point), or that the definitions require the same functional unit type, or that the candidates have the same implication for integer register (GR) and floating-point register (FR) pressure.

3.1.2 Candidate Prioritization

The second step of prematerialization assigns each candidate a priority that determines the order in which we will attempt to transform the candidate live ranges. In general it is always beneficial to prematerialize as many candidates as possible but the number of nops in the code stream can limits the number of prematerializations. Further, some nop slots are more valuable than others for materialization of definitions. Because of this contention for nops among prematerialization candidates, the priority order is important.

The following factors can be considered in calculating the priority for a candidate:

a) If there are uses in the block containing the original definition, the definition must be kept. Otherwise, it will be

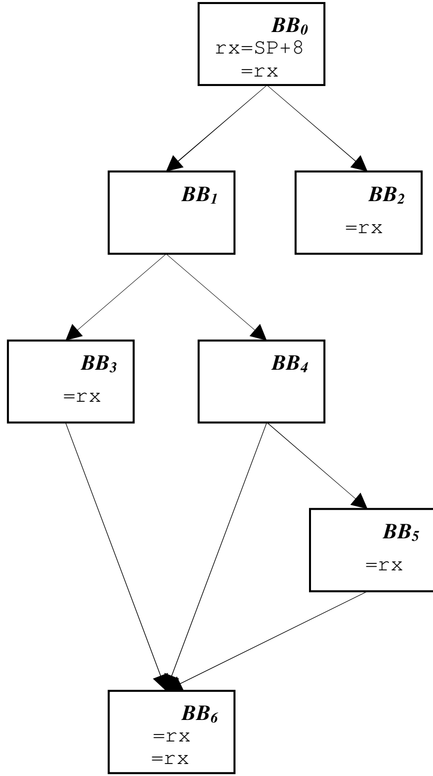

Figure 1. Live range before prematerialization

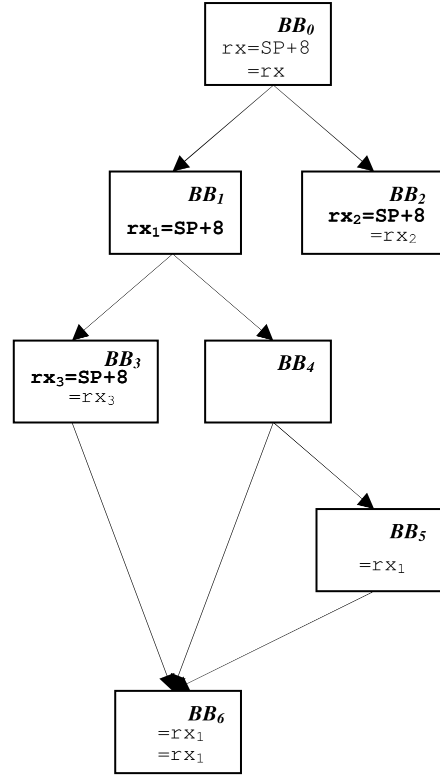

Figure 2. Live range after prematerialization

replaced with a nop if we succeed in finding nops for all uses. Thus, the absence of uses in the definition block may create a new nop that prematerialization can replace with a definition on behalf of a later candidate.

b) A live range that contains uses in *n* basic blocks outside of the block containing the single definition may require up to *n* definitions, and therefore consume up to *n* nops. The more nops are consumed for this candidate, the fewer nops are available for later candidates.

c) The greater is the execution frequency of references to a candidate live range, the more attractive is the candidate. We would prefer to avoid generating spills or fills in frequently executed code.

d) Register pressure could be taken into account by giving priority to live ranges that span basic blocks with a high number of overlapping live ranges. Note that unlike previous factors, register pressure changes with each successful candidate prematerialization.

3.1.3 Definition Materialization

The final step of PM is to materialize additional definitions for each of the candidate live ranges, one candidate at a time, in priority order. Before going into the details of materialization, we first introduce several terms.

A nop instruction is said to be ***suitable*** for a set of uses of a given candidate if (a) all the uses in the set are dominated by the nop, (b) the nop is far enough away in terms of cycles from the uses to satisfy the latency from a copy of the original definition to the uses, and (c) the nop slot must be compatible with the type of the defining instruction.

Note that for a given set of uses of a candidate several suitable nop instructions may exist. However, during the materialization step each set of uses is associated with at most one suitable nop *N*. These uses are said to be ***covered*** by *N*.

For a given candidate, we may perform two kinds of prematerialization: ***full*** prematerialization in which we look for a set of nops that are collectively suitable for all uses of the candidate, and ***partial*** prematerialization in which we look for at least one nop that is suitable for at least one use of the candidate.

Consider a candidate live range whose single definition resides in basic block BB_0. Full prematerialization either completely eliminates the original live range if there are no uses in BB_0, or reduces it to the original definition and the uses in BB_0 otherwise. In contrast, partial prematerialization leaves some other part of the original live range intact, along with the part in BB_0.

The materialization process begins by iterating over all basic blocks with uses other than BB_0. If we fail to find in basic block BB_i a nop suitable for the uses in BB_i, we then attempt to find a suitable nop in the immediate dominator *immDom*(BB_i) of BB_i, and so on, going up the dominator tree rooted at BB_0. Because of the definition in BB_0, copying a new definition there is not beneficial, and so the search terminates after the basic block BB_k such that *immDom*(BB_k) = BB_0. If at any point of the nop search we fail to find a suitable nop for a use – i.e., we reach BB_0 – then

full prematerialization for this candidate is impossible; unless we are willing to perform partial prematerialization, we bail out and move on to the next candidate.

In the search for a nop instruction suitable for uses in BB_i along the path through *immDom*(BB_i) towards the root of the dominator tree we may reach a previously found nop *N* that covers some other set of uses. If *N* is suitable for the uses in BB_i, we stop the search and *N* is employed to cover the uses in BB_i as well. If *N* is not suitable but some other nop *M* in the same basic block as *N* is suitable for the uses in BB_i, as well as those covered by *N*, we stop the search and *M* is employed to cover not only the uses in BB_i but also the uses formerly covered by *N*. Note that because of the way a nop can cover uses in more than one basic block, the order in which we iterate over basic blocks with uses does not affect the final set of nops selected for the candidate.

After we have finished iterating over the basic blocks with uses, we perform prematerialization for this candidate if we have found at least one nop suitable for one use. We replace each selected nop *N* with a copy of the defining instruction that writes a new virtual register, and we rewrite the uses covered by *N* to read that new virtual register. Finally, in the case of full prematerialization, if there are no uses in BB_0 we replace the original definition with a nop.

As an example application of the materialization step for a candidate live range, consider Figure 1 where references to a virtual register *rx* are shown in a control flow graph. We have uses in basic blocks BB_0, BB_2, BB_3, BB_5, and BB_6. When materialization for *rx* is attempted, suppose we visit the non-BB_0 basic blocks with uses in numerical order. The search for suitable nops proceeds as follows. Assuming there is a nop in BB_2 suitable for the use there, we record this nop as covering that use. We process BB_3 similarly, finding a suitable nop in that block and recording it. Moving on to BB_5, suppose there is no suitable nop in that block or in its immediate dominator BB_4, but that in the next dominator BB_1 we find a nop that is suitable for the use in BB_5; we record it as covering that use. Finally, suppose there is no suitable nop in BB_6, so we ascend the dominator tree to BB_1. We find that the nop in BB_1 recorded as covering the use in BB_5 is also suitable for the use in BB_6, so we record this nop as also covering the use in BB_6.

We have successfully found suitable nops for all uses of *rx* and therefore perform full prematerialization for *rx*. A definition of a new register rx_2 is created in BB_2 and the use in BB_2 is rewritten to reference rx_2. We process BB_3 similarly, creating a new register rx_3. Next, we create a definition of a new register rx_1 in BB_1 and rewrite the uses in BB_5 and BB_6. Finally, because there is a use in BB_0, we must retain the original definition in BB_0. The result of the prematerialization of live range *rx* is shown in Figure 2. Note that the original live range is now split into four smaller ones: *rx*, rx_1, rx_2, and rx_3.

3.2 Prematerialization in HP-UX Compilers

This section provides details of how prematerialization is implemented in our compilers. We present the prematerialization classes, priority function, and structure of the prematerialization driver.

3.2.1 Prematerialization Classes and Candidate Prioritization

In our compilers, we divide the candidates for prematerialization into two classes.

The ***Integer class*** consists of live ranges residing in integer registers whose prematerialization reduces GR pressure. The instruction defining a live range of this class is a simple arithmetic instruction that defines a single GR and uses only immediates or certain GRs that are seen by the register allocator as live throughout the entire procedure. Such an instruction typically materializes either an integer literal or part of an address.

Wherever we place new copies of the definition, we do not extend the lifetime of any source register; therefore, prematerializing a live range of this class should always be profitable. Because the latency from any of these defining instructions to a use is generally one cycle [12], we have a lot of freedom in finding suitable nops for live ranges of this class.

The Itanium® ISA provides for moving values from the integer register file to the floating-point register file via the *setf* instruction. The ***Setf class*** consists of live ranges residing in floating-point registers whose prematerialization reduces FR pressure (possibly at the cost of increasing GR pressure). The instruction defining a live range of this class is of the form *setf fx* = *ry* where *ry* has a single defining instruction. Such an instruction is generally used either to materialize a floating-point constant or to prepare for integer multiplication using the *xma* instruction (which executes on a floating-point functional unit).

Prematerializing an instruction of the Setf class is likely to extend the live range of *ry* in order to simplify the live range of *fx*. This tradeoff can be profitable in the presence of a procedure call or a software pipelined loop as explained in Section 2.1. We begin with the assumption that it is profitable in general, and then employ two heuristics to determine whether to suppress prematerialization for the Setf class in a given procedure. First, if this procedure contains no procedure calls and no software pipelined loops, then the fact that the upper 96 FRs are scratch and rotating is irrelevant. Therefore, there is no need for prematerialization of this class. Second, we make a quick estimate of the GR pressure compared to the FR pressure in this procedure. If the relative GR over FR pressure exceeds a specified threshold, then we assume that prematerializing a candidate of the Setf class unnecessarily risks enlarging the integer register stack frame or even incurring GR spill code, and we suppress Setf prematerialization. These two heuristics are integrated into the candidate selection step.

The latency from the definition of *fx* to its use is at least 6 cycles [12], which restricts our ability to find suitable nops for live ranges of the Setf class. Furthermore, because of asymmetries in the processor issue rules for *setf*, it is not sufficient to find a nop that is far enough away from the uses of *fx* and occurs in a bundle with an appropriate template. We need to analyze all instructions scheduled in the cycle containing the nop to determine whether we can replace the nop with a *setf* without causing a dispersal stall, which forces a cycle break before the *setf*. This analysis is integrated into the nop search employed during the materialization step.

We use the following priority function for each candidate *c*:

$p(c) = 2*n + d$

where n is the number of basic blocks containing uses of the live range, excluding the block BB_0 with the single definition. The value of d is 1 if the candidate c contains at least one use in BB_0 and 0 otherwise. Lower priority values indicate more attractive candidates. This priority function prefers candidates which would consume the fewest nop slots for full prematerialization if each use block were to receive its own definition; and, secondarily, prefers candidates for which full prematerialization would eliminate the original definition.

3.2.2 Prematerialization Driver

The prematerialization method is implemented in our compilers as follows.

```
(1) prematerialization() {
(2)   candidates = selectCandidates()
(3)   sortCandidates(candidates)
(4)   fullPM(integerCandidates(candidates))
(5)   partialPM(integerCandidates(candidates))
(6)   fullPM(setfCandidates(candidates))
(7)   partialPM(setfCandidates(candidates))
(8) }
```

Routines `selectCandidates` (line 2) and `sortCandidates` (line 3) implement the candidate selection and prioritization, respectively, and the sequence of `fullPM` and `partialPM` (lines 4 through 7) implements the definition materialization step.

We perform all prematerialization for the Integer class (lines 4 and 5) before any prematerialization for the Setf class (lines 6 and 7) to ensure that Setf prematerialization does not use up nops that could be used for Integer candidates (*setf* is *M*-type, as are some of the defining instructions in the Integer class).

Partial prematerialization can be profitable: although we do not completely break apart the original live range, we do simplify it and thus help the subsequent coloring process. However, using up valuable nops reduces the chance to prematerialize later candidates. For this reason, we perform partial prematerialization for any class only after we have attempted full prematerialization for all candidates of that class.

Because successful full prematerialization of a candidate with no uses in BB_0 opens up a nop slot (previously occupied by the original definition), it is possible that a higher priority candidate that could not undergo full prematerialization for lack of a suitable nop can achieve it if visited again. Therefore, `partialPM` (lines 5 and 7) may in fact perform full PM for a candidate.

4. EXPERIMENTAL EVALUATION

In this section we evaluate the PM implementation in the HP-UX production compilers for the Intel® Itanium® architecture. We perform two separate evaluations: ***IntegerPM*** does prematerialization for the Integer class only, and ***IntegerSetfPM*** does prematerialization for both the Integer and Setf classes. The experiments were run on an Itanium® 2 processor under the HP-UX operating system. The baseline in our experiments was obtained by compiling the SPEC2000 suite of benchmarks [17] with very aggressive switches (+O4 +Ofaster +Oprofile=use) which include all interprocedural, loop, and profile-based optimizations. Our compiler employs a graph coloring-based global register allocator which includes Briggs-style rematerialization at spill time but does not perform any live range splitting or rematerialization prior to register allocation. We consider a variety of measures including traditional memory spills/fills, RSE stalls, and overall execution times, to show that opportunities exist for PM to significantly reduce register pressure.

Table 2. Improvement in execution time due to PM (in %)

SPEC2000	IntegerPM	IntegerSetfPM
164.gzip	*0.7*	*0.7*
176.gcc	*0.6*	*0.6*
186.crafty	*0.6*	*0.6*
197.parser	*0.3*	*0.3*
252.eon	*0.8*	*1.7*
253.perlbmk	*0.2*	*0.2*
254.gap	*1.0*	*1.0*
255.vortex	*2.1*	*2.1*
300.twolf	*0.2*	*0.4*
168.wupwise	*1.3*	*2.9*
173.applu	*3.3*	*18.7*
177.mesa	*1.9*	*3.6*
191.fma3d	*0.1*	*6.1*

4.1 Run-Time Performance

Both IntegerPM and IntegerSetfPM improve the overall performance of most benchmarks in the SPECint suite and four benchmarks in the SPECfp suite (168.wupwuse, 173.applu, 177.mesa, and 191.fma3d). Table 2 shows the percentage improvement with prematerialization enabled compared to the base case without PM. There is no impact, positive or negative, on 175.vpr, 181.mcf, or 256.bzip2 in SPECint, or on the remaining benchmarks in SPECfp. Execution time differences greater then or equal to 0.2% are considered significant and reproducible in our testing environment.

Given the prevalence of integer computation, IntegerSetfPM does not further improve upon IntegerPM on most SPECint applications. The only exception is 252.eon where Setf prematerialization accounts for half the total gain achieved by IntegerSetfPM. This interesting result will be explained by the distribution of memory spills/fills presented in the next section. However, IntegerSetfPM has an outstanding impact on the selected floating-point applications, speeding up 173.applu and 191.mesa by 18.7% and 6.1%, respectively. The observed performance gains result from reduced register pressure due to prematerialization. We now analyze register pressure according to its two manifestations on Itanium®: explicit and RSE spill and fills.

4.2 Explicit Spills and Fills

We instrumented the register allocator to report traditional memory spills, fills, and rematerializations introduced by its spill

Table 3. GR spill data for SPECint

benchmark	spills	fills	remats	dynamic spills	dynamic fills	dynamic remats
164.gzip	15/8	32/8	83/29	1.4e5/2.6e4(81%)	6.0e5/2.1e5(66%)	2.3e5/4.0e4(83%)
175.vpr	25/21	43/33	109/10	9.6e2/1.4e2(85%)	8.2e3/2.6e3(69%)	9.2e2/2.3e2(75%)
176.gcc	79/67	111/75	164/24	2.4e4/2.1e4(13%)	5.8e4/3.9e4(33%)	9.5e3/9.4e2(90%)
186.crafty	10/7	36/26	24/0	1.8e2/1.4e2(23%)	1.3e3/4.5e2(65%)	1.6e16/0(100%)
197.parser	9/1	19/1	62/0	6.3e1/7.0e0(89%)	2.0e4/2.2e3(89%)	8.9e3/0(100%)
252.eon	115/116	177/176	87/47	1.8e3/1.8e3(-1%)	5.3e3/5.3e3(0%)	3.6e1/6.0e0(83%)
254.gap	12/9	12/8	0/0	2.3e5/8.2e4(65%)	2.5e5/9.3e4(62%)	0/0
300.twolf	114/80	164/101	212/43	7.0e7/3.8e7(46%)	1.7e8/1.3e8(25%)	1.5e8/6.0e5(100%)

Table 4. FR spill data for SPECint

benchmark	spills	fills	remats	dynamic spills	dynamic fills	dynamic remats
175.vpr	43/43	43/43	0/0	1.5e7/1.5e7(0%)	1.5e7/1.5e7(0%)	0/0
252.eon [IntegerPM]	22/22	40/40	4/4	5.8e7/5.8e7(0%)	1.7e8/1.7e8(0%)	1.8e4/1.8e4(0%)
[IntegerSetfPM]	22/21	40/40	4/4	5.8e7/4.0e7(31%)	1.7e8/1.3e8(23%)	1.8e4/1.8e4(0%)
300.twolf	8/8	15/15	0/0	4.8e3/4.8e3(0%)	1.1e4/1.1e4(0%)	0/0

Table 5. GR spill data for SPECfp

benchmark	spills	fills	remats	dynamic spills	dynamic fills	dynamic remats
173.applu [IntegerPM]	236/187	606/516	283/243	1.6e6/1.2e6(26%)	1.8e7/6.0e6(67%)	1.6e6/3.4e4(98%)
[IntegerSetfPM]	236/250	606/657	283/334	1.6e6/1.6e6(2%)	1.8e7/1.2e7(35%)	1.6e6/2.7e6(-64%)
177.mesa [IntegerPM]	33/32	63/60	19/19	1.4e7/1.3e7(4%)	7.4e7/6.6e7(11%)	3.9e6/2.9e6(24%)
[IntegerSetfPM]	33/32	63/60	19/26	1.4e7/1.3e7(4%)	7.4e7/6.6e7(11%)	3.9e6/3.6e6(6%)
191.fma3d [IntegerPM]	237/177	397/294	1090/480	1.4e9/7.1e8(50%)	1.6e10/1.1e10(29%)	6.7e9/2.1e9(69%)
[IntegersetfPM]	237/178	397/296	1090/535	1.4e9/7.2e8(49%)	1.6e10/1.1e10(29%)	6.7e9/3.5e9(48%)

Table 6. FR spill data for SPECfp

benchmark	spills	fills	remats	dynamic spills	dynamic fills	dynamic remats
168.wupwise [IntegerPM]	7/7	19/19	0/0	5.7e6/5.7e6(0%)	1.2e11/1.2e11(0%)	0/0
[IntegerSetfPM]	7/4	19/9	0/0	5.7e6/3.2e6(45%)	1.2e11/9.9e6(100%)	0/0
173.applu [IntegerPM]	219/219	742/742	0/0	4.5e8/4.5e8(0%)	2.9e9/2.9e9(0%)	0/0
[IntegerSetfPM]	219/201	742/451	0/0	4.5e8/3.4e8(25%)	2.9e9/9.9e8(66%)	0/0
177.mesa [IntegerPM]	106/106	256/256	0/0	3.2e10/3.2e10(0%)	3.5e11/3.5e11(0%)	0/0
[IntegerSetfPM]	106/93	256/204	0/0	3.2e10/2.0e10(38%)	3.5e11/1.9e11(45%)	0/0
191.fma3d [IntegerPM]	685/685	1283/1283	11/11	5.3e11/5.3e11(0%)	2.8e12/2.8e12(0%)	4.9e6/4.9e6(0%)
[IntegersetfPM]	685/601	1283/1053	11/8	5.3e11/3.0e11(43%)	2.8e12/1.1e12(62%)	4.9e6/4.9e6(0%)

component. Each of the Tables 3 through 6 presents the data for a given register file and benchmark suite; for each benchmark, it shows the number of static spills, fills, and rematerializations (remats), followed by the corresponding dynamic data. A static figure gives the number of operations of the specified type generated for the benchmark. A dynamic figure gives the sum of execution frequencies of operations of the specified type for the benchmark, where the execution frequencies are obtained by running the benchmark with training input(s). The first number in each pair gives the data for the compiler with the PM phase disabled, and the second number gives data for the compiler with the PM phase enabled. After each dynamic pair we also show the percentage improvement (reduction in the number of executed operations of the specified type) due to PM.

Tables 3 and 4 show spill related data for GRs and FRs, respectively, for the SPECint suite of benchmarks. There is no data for 181.mcf, 253.perlbmk, 255.vortex, and 256.bzip2 because in these benchmarks the compiler does not generate spill related code. In Table 3 we do not distinguish between IntegerPM and IntegerSetfPM because they behave the same for the benchmarks in this table.

The results in Table 3 indicate that prematerialization is very effective in reducing the amount of GR spill related code for the

Table 7. RSE contribution to execution time (in %)

SPECint2000	no PM	IntegerSetfPM	SPECfp2000	no PM	IntegerSetfPM
164.gzip	*0.1*	*0.1*	168.wupwise	*0.1*	*0.1*
175.vpr	*0.2*	*0.2*	171.swim, 172.mgrid	*0.0*	*0.0*
176.gcc	*4.1*	*3.8*	173.applu	*0.0*	*0.0*
181.mcf	*0.1*	*0.1*	177.mesa	*8.0*	*7.3*
186.crafty	*8.2*	*7.1*	178.galgel	*0.1*	*0.1*
197.parser	*2.4*	*2.3*	179.art	*0.0*	*0.0*
252.eon	*2.9*	*2.6*	183.equake	*0.1*	*0.1*
253.perlbmk	*3.0*	*2.8*	187.facerec	*0.1*	*0.0*
254.gap	*2.6*	*1.6*	188.ammp, 189.lucas	*0.0*	*0.0*
255.vortex	*5.7*	*4.2*	191.fma3d	*0.4*	*0.4*
256.bzip2	*1.1*	*1.0*	200.sixtrack	*0.2*	*0.2*
300.twolf	*0.2*	*0.2*	301.apsi	*1.1*	*1.0*

SPECint applications: on average PM reduces the number of dynamic spills and fills by at least half, and it almost completely eliminates the need for rematerialization at spill time.

Several SPECint benchmarks also exhibit floating-point pressure as shown in Table 4. Although Setf prematerialization was chiefly designed and tuned for FP intensive applications, it does help in 252.eon by reducing the number of dynamic FR spills and fills by 31% and 23%, respectively.

Tables 5 and 6 show spill related data for GRs and FRs, respectively, for a subset of the SPECfp suite of benchmarks. We present data for 168.wupwuse, 173.applu, 177.mesa, and 191.fma3d, where our prematerialization techniques have a measurable effect. There is no row for 168.wupwise in Table 5 because we do not spill or rematerialize any GRs in this application. Unlike the SPECint benchmarks, the selected SPECfp benchmarks behave differently for IntegerPM and IntegerSetfPM, and therefore we report data for both PM experiments.

Note in Table 5 that IntegerSetfPM is less successful than IntegerPM in reducing the GR related spills, fills, and remats, due to its additional Setf prematerialization. Each successful prematerialization of a *setf* candidate simplifies an FR live range, usually at the expense of complicating a GR live range. Over the four benchmarks IntegerSetfPM reduces the number of dynamic floating-point spills and fills by an arithmetic mean of 38% and 68%, respectively, which far outweighs the lost integer prematerialization opportunities. It is interesting to note that on 173.applu IntegerSetfPM degrades the most for GR spills/fills compared to IntegerPM, and the same time substantially improves FR spill/fills. However, the net effect of this GR/FR tradeoff is an amazing 18.7% execution speedup for this benchmark.

The performance improvement from the reduction in explicit spills/fills cannot be determined directly. First, the dynamic figures only approximate the actual number of run time spill operations, because of the difference between benchmark training and reference inputs. Second, each "operation" accounted for above may consist of multiple instructions, e.g. address materialization plus memory access. Third, the additional memory traffic from spills and fills may perturb the application's memory traffic in unknown ways. Finally, register allocation is followed by several other phases, including the rescheduler, which attempts to improve the final schedule of code containing spill related sequences. See the following section for further discussion of performance improvement attribution.

4.3 RSE Spills and Fills

In addition to the explicit spill code, register stack engine operations are another, unique for Itanium® processors, manifestation of register pressure. The amount of time that the processor is stalled waiting for a register window to spill/fill can be measured via the Itanium® PMU [12]. Unfortunately, the second order effects of these spills and fills in the memory hierarchy on subsequent memory operations issued by the application cannot be accurately measured. Table 7 presents the measurable percentage contribution of the RSE to the total execution time without PM and with IntegerSetfPM. This represents a lower bound on the percentage execution time attributable to the RSE. Notice that we observe a greater reduction in RSE traffic for the SPECint applications, which is to be expected because the RSE operates only on the integer register file.

One should consider the data in Tables 3 through 6 in conjunction with the data in Table 7 to correlate the execution time improvements with the reduction of register pressure.

For example, Table 2 indicates a 0.7% overall improvement in 164.gzip. Because PM does not change the RSE contribution to execution time for this benchmark (Table 7), the most of the improvement must come from reducing explicit dynamic spills by 81% and fills by 66% as shown in Table 3 for 164.gzip.

Consider now the data for 255.vortex. Table 2 indicates a 2.1% overall improvement and the majority of this improvement can be directly attributed to reducing the RSE stall contribution to execution time from 5.7% to 4.2% as shown in Table 7. Because the compiler does not generate spill code for this benchmark, the

remaining 0.7% improvement (2.1 – (5.7-4.2)) is attributed to second order effects caused by the RSE.

Determining the reason for the performance improvement of 0.6% in 176.gcc is more difficult. The 0.6% improvement exceeds the 0.3% reduction in the RSE stall contribution (Table 7); the remaining 0.3% is presumably due to a combination of the 13% and 33% reduction in the explicit dynamic spills and fills, respectively (Table 3), and indirect effects of the RSE.

Finally, note that even in the absence of spill related code and RSE stalls, prematerialization can reduce the number of saved callee-saved/preserved registers - and therefore the amount of register save and restore code in procedure prologs and epilogs - and thereby still benefit performance.

5. DISCUSSION

We have presented prematerialization, a novel method for reducing register pressure for VLIW architectures. Prematerialization takes advantage of the fact that ILP is not evenly distributed throughout applications and makes use of nops as insertion points for copies of defining instructions. By placing definitions into nop slots, prematerialization provides a mechanism to reduce register pressure without any increase in path length due to stalls or increase in the number of instructions executed. This approach provides for a significant reduction in both traditional spill/fill code and in the contribution to runtime from the Itanium® register stack engine.

Although our PM implementation targets Itanium® processors, the method can be employed in any compiler for a VLIW processor with nop instructions. For a variety of reasons, including alignment requirements, instruction issue restrictions, and stall enforcement, nops occur explicitly in a VLIW binary or are inserted into the long words when the instruction enters the instruction cache [8]. Despite advanced techniques to expose ILP, even the best compilers cannot produce nop-free schedules. Given that embedded VLIW processors usually have fewer registers than Itanium® processors, it would be particularly interesting to evaluate the impact of prematerialization in such an environment.

Our PM implementation runs prior to register allocation. An alternative would be to perform PM after building the interference graph, i.e., in the core of register allocation. This approach would require complicated and expensive incremental updates of dataflow analysis results and the interference graph; however, at this point we would have detailed information about the contribution of a live range to the overall register pressure, and we might be able to use this information to better guide prioritization of candidates. We could also enhance Briggs-style rematerialization with spill-time prematerialization; in addition to the aforementioned issues with PM during register allocation, this approach would also require changes to spill weights to begin spilling with prematerializable live ranges.

It is possible that PM could result in increased path length due to unlucky interaction with spill code generation and post register allocation rescheduling. However, we have not observed performance loss from this problem.

There are a number of avenues that we would like to explore in the future. The candidate priority function can probably be improved by taking into account the execution frequency of references to a live range, or knowledge of register pressure, as described in Section 3.1.2. Our current implementation requires that we find a compatible nop for each defining instruction. A more aggressive approach might try to alter the template for a bundle containing an incompatible nop (or perhaps the templates for some number of adjacent bundles) and potentially reorder the instructions in order to create a nop slot of the necessary type. We would like to expand the set of live ranges candidates for prematerialization – e.g., pure floating-point instructions, or chains of dependent instructions. Finally, placing copy instructions in nop slots to split selected live ranges and reduce register pressure also merits investigation.

6. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments to improve the presentation.

7. REFERENCES

[1] Berson, D., Gupta, R., and Soffa, M. URSA: a unified resource allocator for registers and functional units in VLIW architectures. *Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, 1993.

[2] Briggs, P. *Register Allocation via Graph Coloring*. Ph.D. Thesis, Rice University, Houston, TX, April 1992.

[3] Briggs, P., Cooper, K., and Torczon, L. Rematerialization. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[4] Briggs, P., Cooper, K., and Torczon, L. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428-455, May 1994.

[5] Chaitin, G. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):98-105, June 1982.

[6] Govindarajan, R., Yang, H., Amaral, J., Zhang, C., and Gao, G. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Transactions on Computers*, 52(1):4-20, January 2003.

[7] Gupta, R. and Bodik, R. Register pressure sensitive redundancy elimination. *Proceedings of International Conference on Compiler Construction (CC99)*, LNCS, vol. 1175, Springer-Verlag, 1999.

[8] Haga, S., Zhang, Y., Webber, A., and Barua, R. Reducing code size in VLIW instruction scheduling. *Journal of Embedded Computing*, 1(4), 2005.

[9] Hank, R. *Region-based Compilation*. Ph.D. Thesis, University of Illinois, Urbana, IL, May 1996.

[10] Hoflehner, G., Kirkegaard, K., Skinner, R., Lavery, D., Lee, Y., and Li, W. Compiler optimizations for transaction processing workloads on Itanium Linux systems. *Proceedings of 37th Annual International Symposium on Microarchitecture*, 2004.

[11] Intel, Intel® Itanium® Architecture Software Developer's Manual Vol. 1-3, January 2006.

[12] Intel, Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization, 2002.

[13] Intel, Intel® Itanium® Software Conventions & Runtime Architecture Guide, 2002.

[14] Lowney, P., Freudenberger, S., Karzes, T., Lichtenstein, W., Nix, R., O'Donnell, J., and Ruttenberg, J. The Multiflow trace scheduling compiler. *The Journal of Supercomputing* 7 (1993): 51—142.

[15] Settle, A., Connors, D., Hoflehner, G., and Lavery, D. Optimization for the Intel Itanium architecture register stack. *Proceedings of International Symposium on Code Generation and Optimization*, 2003.

[16] Simpson, L. *Value-driven redundancy elimination*. Ph.D. Thesis, Rice University, Houston, TX, May 1996.

[17] Standard Performance Evaluation Corporation, http://www.spec.org.

[18] Touati, S. Register saturation in superscalar and VLIW codes. *Proceedings of International Conference on Compiler Construction (CC01)*, LNCS, vol. 2027, Springer-Verlag, 2001.

[19] Zhao, M., Childers, B., and Soffa, M. Model-based framework: an approach for profit-driven optimization. *Proceedings of International Symposium on Code Generation and Optimization*, 2005.