Overcoming Static Register Pressure for Software Pipelining in the Itanium Architecture*

Haibo Lin, Wenlong Li, and Zhizhong Tang

Tsinghua University,
Beijing 100084, P.R. China
{linhaibo99,liwenlong00}@mails.tsinghua.edu.cn
tzz-dcs@tsinghua.edu.cn

Abstract. Software pipelining techniques have been shown to significantly improve the performance of loop-intensive scientific programs. The Itanium architecture contains many features to enhance parallel execution, such as support for efficient software pipelining of loops. The drawback of software pipelining is the high register requirements, which may lead to software pipelining failure due to limited static general registers in Itanium. This paper evaluates the register requirements of software-pipelined loops. It then presents a novel register allocation scheme, which allocates stacked registers to serve as static registers. Experimental results show that this method gains an average 2.4% improvement, and a peak 18% improvement in performance on NAS Benchmarks.

1 Introduction

The Itanium architecture contains many features to enhance parallel execution, such as an explicitly parallel (EPIC) instruction set, large register file, etc. It also provides features such as register rotation [1] to support efficient software pipelining without increasing the code size of the loop. Software pipelining [2] tries to improve the performance of a loop by overlapping the execution of several successive iterations. This improves the utilization of available hardware resources by increasing the instruction level parallelism (ILP).

The drawback of aggressive scheduling techniques [3] such as software pipelining is that they increase register requirements [4]. There have been proposals to perform register allocation for software-pipelined loops [5]. If the number of registers required is larger than the available number of registers, spill code has to be introduced to reduce register usage, or software pipelining is given up.

This paper evaluates static register requirements of software-pipelined floating-point intensive loops. A new method to software pipeline loops that require more static general registers than those Itanium provides is also presented. It uses stacked general register to serve as static general register, thus increasing software-pipelined loops that require many static general registers but few rotating general registers. It has been implemented in Open Research Compiler (ORC), and results in an average

* This work was supported by NSFC grant 60173010.

X. Zhou et al. (Eds.): APPT 2003, LNCS 2834, pp. 109–113, 2003.

2.4% improvement and a peak 18% improvement in performance on NAS Benchmarks.

2 Itanium Architecture Features

Itanium provides 128 general registers, 128 floating-point registers, and 64 predicate registers. The general registers are partitioned into two subsets. GR0-GR31 is termed the static general register. GR32-GR127 is termed the stacked general register. The stacked registers are made available to a program by allocating a register stack frame consisting of a programmable number of local and output register. The floating-point registers and predicate registers are also partitioned into two subsets respectively. FR0-FR31 (PR0-PR15) is termed the static floating-point (predicate) register. FR32-FR127 (PR16-PR63) is termed the rotating floating-point (predicate) register.

A fixed-size area of the floating-point and predicate register files(FR32-FR127 and PR16-PR63), and a programmable-sized area of the general register file are defined to rotate. The size of the rotating area in the general register file is determined by an immediate in the `alloc` instruction and must be either zero or a multiple of 8, up to a maximum of 96 registers. The lowest numbered rotating register in the general register file is GR32.

3 Register Requirements of Software Pipelining in the Itanium Architecture

This section evaluates static general register requirements of all the innermost loops of the NAS Benchmarks that are suitable for software pipelining in the Itanium architecture. These loops have been obtained with the ORC compiler. A total of 363 loops amenable for software pipelining have been used. This set includes all the innermost loops that do not have subroutine calls or conditional exits. Loops with conditional structures in their bodies have been IF-converted, with the result that the loop now looks like a single basic block.

In the Itanium architecture, three kinds of variables require static general registers. They are:

1. Dedicated variants, which require special general registers such as global pointer (gp), memory stack pointer (sp);
2. Loop invariants, the values of which are repeatedly used by a loop at each iteration, but never written by it;
3. Base update variants, something like induction variables. Consider the following load instruction `ld4 r1=[r2], 4`. After the value consisting of 4 bytes is read from memory starting at the address specified by the value in GR r2, the value in GR r2 is added to 4, and the result is placed back in GR r2. The variable in GR r2 is termed base update variant. Although the value of such a variable varies at each iteration, the def and ref are involved in one instruction, resulting in a live-range of 0 cycle. So we should assign it a static general register instead of rotating general register.

Figure 1 shows the cumulative distribution of the requirements for static general registers for all 363 loops. In general, loops have very few dedicated variants. For instance 96% of the loops have no dedicated variants and only 5 loops have 1 dedicated variant. Loops also have few loop invariants. For instance 65% of the loops have no loop invariants and the other loops have at most 6 invariants. Nevertheless loops have a high number of base update variants. For instance 3 loops have more that 32 base update variants.

Taking into account of several special registers, such as global pointer (gp), memory stack pointer (sp), reserved thread pointer (tp), and 1 or 2 register serving other purposes, there are only 27-28 out of 32 static general registers available for software pipelining. Since 6% of the loops require such many static general registers, these loops will fail in software pipelining phase.

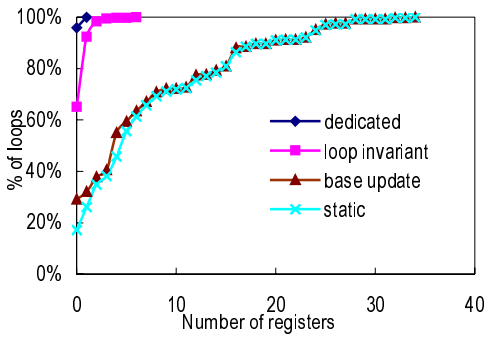

Fig. 1. Cumulative distribution of variants requiring static general registers. Each point (x,y) of the graph represents the percentage y of loops having less than x registers

4 Stacked Register Allocation

General registers GR32-GR127 form a register stack that is automatically managed across procedure calls and returns. Each procedure frame on the register stack is divided into two dynamically sized regions, one for input parameters and local variables, and one for output parameters. The hardware makes no distinction between input and local registers. On a procedure call, the registers are automatically renamed by the hardware so that the caller's output registers form the base of the callee's new register stack frame. On return, the registers are restored to the previous state, so that the input and local registers are preserved across the call.

A subset of the registers in the procedure frame may be designated as rotating registers. The rotating register region always starts with GR32, and may be any multiple of 8 registers in number, up to a maximum of 96 rotating register. The renaming is under control of the Register Rename Base (RRB). If the rotating registers include any or all of the output register, software must be careful when using the output registers for passing parameters, since a non-zero RRB will change the virtual register numbers that are part of the output region. In general, software should either ensure that the rotating region does not overlap the output region, or that the RRB is cleared to zero before setting output parameter registers.

Since the rotating general register requirements of software pipelining are not very high, there are often some unused general registers in the register stack. We propose a novel register allocation scheme for software pipelining called Stacked Register Allocation (SRA). SRA tries to allocate these registers to variants needing static general registers. Actually, SRA allocates stacked non-rotating registers right on top of stacked rotating registers, and put the rest of the register stack frame to higher register regions. SRA realizes dynamic partition between static general registers and rotating general registers to some extent (SRA can not reduce the number of static general registers to less than 32). It improves the utilization efficiency of general registers, and results in more software-pipelined loops. Fig. 2 shows register stack usage model before and after SRA.

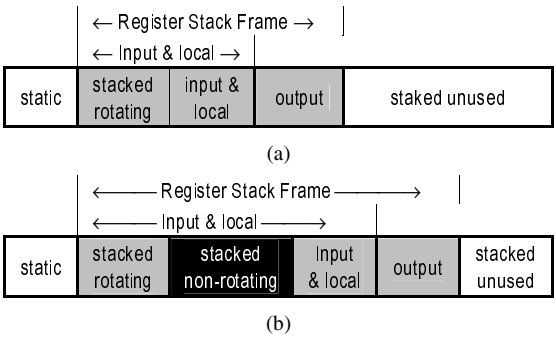

Fig. 2. Register stack usage model: (a) before SRA, (b) after SRA

5 Experimental Results

We have implemented SRA in the ORC compiler for Itanium. In this section, we compare the results of using this technique and original implementation in ORC on the NAS suite of benchmark programs, that is, the speedup of our method. These results were obtained by running the benchmarks with -O3 level optimization. Profile feedback and Inter-Procedural Analysis (IPA) were not used.

Provided with 32 physical general registers, the number of loops that fail in software pipelining caused by non-enough static general registers range from 0 to 11. These loops are all pipelined after SRA is applied. It is reasonable that the number of un-pipelined loops increases when the number of available general registers decreases. SRA can solve the problem of software pipelining failure even in the context of 24 general registers.

Figure 3 shows the percentage improvements of SRA in execution times on an Itanium 733 MHz machine. SRA results in an average 2.4% improvement and a peak 18% improvement in performance with 32 general registers. This result is rather exciting because only 6% of loops are optimized. When the number of available general registers reduces to 24, SRA gains an average speedup of 3.1%.

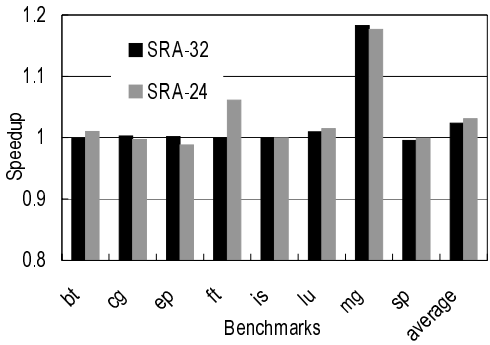

Fig. 3. Performance of SRA. SRA-32 and SRA-24 refer to the number of available static general registers of 32 and 24 respectively

6 Conclusion

In this paper we have evaluated the static general register requirements of software-pipelined loops of NAS Benchmarks on Itanium architecture. We have also shown that some loops with high static general register requirements fail in software pipelining phase. A new method is presented to increase software-pipelined loops on Itanium architecture. It allocates free general registers from register stack for loops that require more static general registers than those Itanium provides. The experimental results show significant improvements in the execution time of NAS Benchmarks.

References

1. Dehnert J. C., Hsu P. Y., Bratt J. P.: Overlapped Loop Support in the Cydra 5. Proceedings of ASPLOS'89. (1989) 26–38
2. Allan V. H., Jones R. B., Lee R. M., Allan S. J.: Software Pipelining. ACM Computing Surveys. **27** (1995) 367–432
3. Huff R. A.: Lifetime-sensitive modulo scheduling. Proceedings of PLDI'93. (1993) 58–267
4. Llosa J.: Reducing the Impact of Register Pressure on Software Pipelining. PhD thesis. Universitat Politècnica de Catalunya (1996)
5. Rau B. R., Lee M., Tirumalai P., Schlansker P.: Register allocation for software pipelined loops. Proceedings of PLDI'92. (1992) 283–299