Inter-Procedural Stacked Register Allocation for Itanium® Like Architecture

Liu Yang[*] Sun Chan[**] G. R. Gao[***] Roy Ju[**] Guei-Yuan Lueh[**] Zhaoqing Zhang[*]

*Institute of Computing Technology, CAS, Beijing 100080,P.R.China

{ly, zqzhang}@ict.ac.cn,

[**]Microprocessor Research Labs, Intel Labs,Santa Clara,CA 95052,USA

{sun.c.chan, roy.ju, guei-yuan.lueh}@intel.com,

[***] Department of Electrical and Computer Engineering, University of Delaware, USA

ggao@udel.edu

ABSTRACT

A hardware managed register stack, Register Stack Engine (RSE), is implemented in Itanium® architecture to provide a unified and flexible register structure to software. The compiler allocates each procedure a register stack frame with its size explicitly specified using an *alloc* instruction. When the total number of registers used by the procedures on the call stack exceeds the number of physical registers, RSE performs automatically register overflows and fills to ensure that the current procedure has its requested registers available. The virtual register stack frames and RSE alleviate the need of explicit spills by the compiler, but our experimental results indicate that a trade-off exists between using stacked registers and explicit spills under high register pressure due to the uneven cost between them. In this work, we introduce the stacked register quota assignment problem based on the observation that reducing stacked register usage in some procedures could reduce the total memory access time of spilling registers, which includes the time caused by the loads/stores due to explicit register spills and RSE overflows/fills. We propose a new inter-procedural algorithm to solve the problem by allocating stacked registers across procedures based on a quantitative cost model. The results show that our approach can improve performance significantly for the programs with high RSE overflow cost, e.g. **perlbmk** and **crafty**, improved by 14% and 3.7%, respectively.

Categories and Subject Descriptors

D3.4 [Programming Languages]: Processors – Compiler; Optimization;

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

Inter-procedural Stacked Register Allocation, Quota Assignment, Hotspot, Hot Region, Register Allocation

1 INTRODUCTION

The register allocation is to minimize the memory access time by choosing a subset of all variables that can be placed in registers and assigning a register to every variable in the subset. Prior researchers have studied register allocation extensively in the literature [1][2][3][4][5]. When a variable is not allocated to a register, it is spilled to memory and explicit load and store instructions are inserted to fetch and store its value, respectively. In addition to the spills due to the limited number of registers, there are also many spills (i.e. save and restore scratch registers) inserted due to the variables live across call sites. Different approaches [5][6][7][8][9] have been proposed to eliminate some spills around call sites.

The register file of Itanium® architecture contains 128 integer, 128 floating-point, 64 predicate, and 8 branch registers. As an alternative to traditional register files, the integer registers are divided into two areas: r0-r31 are static registers, which are visible to all procedures, and r32-r127 are stacked registers, which are visible only to a given procedure. A register stack frame is the set of stacked registers allocated to a specific procedure. Every procedure starts by allocating a register frame using the *alloc* instruction. When the total size of procedure call stack exceeds the capacity of physical stacked registers, RSE ensures that the requested number of registers is available to the allocation request by saving physical registers into the backing store in memory [10] without explicit program intervention. The saved registers are called overflowed registers. As a procedure returns, its corresponding register stack frame is popped and RSE restores the caller's previous-saved registers from the backing store area. The restore operations are called register

fills.

Each register has an associated cost. For permanent registers (callee-save), compilers save and restore the registers at prolog and epilog, respectively. For scratch registers (caller-save), compilers save and restore the registers across call sites. Compilers manage these two kinds of registers *explicitly*. RSE manages stacked register overflow/fill without explicit program intervention. When a register stack overflow occurs, RSE stalls the program execution to wait for the completion of saving overflowed registers. Likewise, RSE fills also stall the execution. The cost associated with stacked registers overflow is *implicit* because the cost only shows up when register stack overflow/fill happens — the total size of procedure call stack exceeds the capacity of physical stacked registers. Using more stacked registers decreases spill overhead (spill variables to memory) at the expense of potential increases of register stack overflow and fill.

Without modeling the cost of stacked register overflow in register allocation, the compiler perceives a simplistic view that using stacked registers is at no cost. Namely, the compiler holds only partial view about the overhead of register allocation. As a result, the compiler tends to maximize the usage of stacked registers so as to minimize the spill overhead. This simple view is adequate if the overhead of register stack overflow and fill is low. However, the view becomes inadequate for call intensive programs that have deep call chains because they are more likely to incur more register stack overflows and fills. Our experimental study shows that **perlbmk** and **crafty** of SPEC2000Int exhibit 23.8% and 4.8% of RSE cost in overall execution time, respectively.

To minimize the total spill-to-memory cost, we need to choose the right storage classes for variables. It is not always profitable to keep a variable in a register than memory. We propose a new inter-procedural algorithm that balances assigning stacked registers to variables and spilling variables to memory. The algorithm has the following main features:

(1) It uses an *inter-procedural framework* to allocate a register stack frame for each procedure. Specifically, we use a 3-step process. Step-I estimates an intra-procedure register usage. Step-II performs an inter-procedural analysis to assign quota of stacked register usage for every procedure with the goal of minimizing the total number of spills to memory for the whole program. Step-III completes the actual register allocation for each procedure with its given register quota.

(2) Our method is based on a quantitative cost model of invocation frequencies of function calls and the cost of memory accesses due to spills and RSE traffic. We use a *weighted call graph* – i.e. a call graph where each edge is annotated with the invocation frequency obtained from profiling feedback.

(3) Our algorithm traverses the call graph by identifying the hot regions according to the maximum accumulated RSE overflow/fill costs of the regions. The hot regions are analyzed one at a time, and stacked registers are allocated to the procedures within a region. During this process, the register demand might be underestimated during step-I comparing to what may be actually needed. Insufficient register quota may result in additional register spills during step-III. Hence, although the algorithm has the merits of taking the advantage of profiling information as well as leveraging inter-procedural framework to obtain a whole-program view of register usage, its effectiveness has to be carefully evaluated.

We implemented the approach in the Open Research Compiler (ORC) [11] and evaluated its effectiveness on Itanium® systems using the SPEC2000Int programs. We have made the following key observations.

(1) Our algorithm is quite effective in improving execution performance, up to 14%, for the benchmarks in Spec2000Int that have high RSE overhead.

(2) For those benchmarks without obvious performance improvement, our algorithm shows no performance degradation except one with slight degradation of 1.2%.

(3) More concrete analysis shows that the usage of stacked registers in some call-intensive procedures is reduced and as such leads to the reduction of the RSE cost.

(4) Our algorithm appears to be effective in finding out hot regions causing most RSE cost in these benchmarks.

2 BACKGROUND ON REGISTER STACK ENGINE

General registers r32 to r127 form a register stack that operates as a circular buffer containing the most recent created frames [10]. A procedure can allocate a register frame on the register stack using *alloc* instruction if the procedure needs to store values in the stacked registers. Each register stack frame includes four parts: 1) input registers, 2) local registers, 3)

output registers, and 4) rotating registers. Input and rotating are included as part of the local area. The size of a register stack frame is local+output, which can be up to 96 registers, specified by an *alloc* instruction as the following format:

alloc <target_reg>=ar.pfs in,local,out,rot

where the ar.pfs is the Previous Function State register to restore the previous state upon return. RSE is responsible for mapping a register stack frame to a set of physical registers. This process is done by the hardware and transparent to the compiler. When a procedure is called, the stacked registers are renamed such that the caller's first register in the output area becomes r32 for the callee. That is, the frame of the caller and callee overlaps — the output area of the caller is the input area of the callee. When the callee returns, the register renaming is restored to the caller's configuration. This mechanism preserves the stacked registers of the caller without storing them to memory. RSE manages register stack overflows and fills automatically, transparent to application software.

3. MOTIVATING EXAMPLE AND PROBLEM FORMULATION

In this section we introduce the cost model for RSE overflows and fills. We also present a motivating example and formulate the problem to minimize the total spill-to-memory cost.

3.1 Cost of RSE Overflows/Fills

An overflowed register stack frame due to a procedure call will be filled upon return from the procedure. Since programs do not usually exit in the middle of a call path, we consider that the overflow and fill of a register stack frame always come in pair. Even in the case of setjmp/longjmp, a register stack frame is part of architectural state, and the overflow and fill of the register stack frame still occur in pair.

Here is an experiment that studies the RSE cost in the Itanium® processors to overflow and fill each stacked register. The program used in the experiment includes four procedures A, B, C, and D, where A calls B, B calls C, and C calls D. Procedures A, B, and C together use up all of the 96 stacked registers. Therefore, the number of overflows/fills equals to the number of stacked registers used by procedure D.

In this experiment, the size of register stack frame allocated by procedure D varies from 1 to 96 in an increment of 1, and we measure the RSE cost in CPU cycles using *pfmon* – a tool used to read the performance monitors in Itanium® processors and report the dynamic behaviors of program execution. The result is shown in **Figure 1**. We can see that the RSE cost increases linearly with the increase of stacked registers allocated by procedure D. Each *stacked register* overflow/fill costs about 1 cycle except a big gap from 56 to 57, which we have not fully understood the reason yet. Therefore, we could assume that the RSE cost is a linear function of the number of register stack overflows/fills.

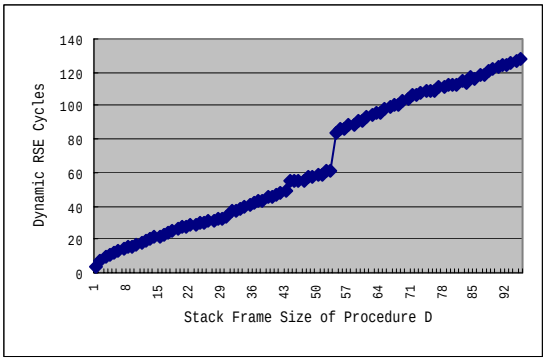

Figure 1. Dynamic RSE Cost vs. Stack Frame Size in Procedure D

3.2 A Motivating Example

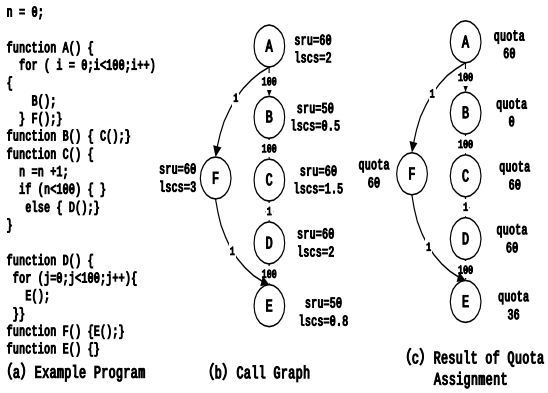

Figure 2. A Motivating Example

Let us illustrate the inter-procedural quota assignment of stacked register usage problem (step-II of our method – see introduction) based on the Itanium® architecture with a simple program P as shown in **Figure 2(a)**. In Itanium® processors, the upper bound of stacked registers is 96. Later in this paper, we will use variable *bound* to represent the upper bound of stacked registers. The following are obtained for each procedure from our method: (1) *stacked registers usage (sru)* and (2) the *load/store operation cycles saved (lscs)* by a given stacked register usage. These can be calculated by an intra-procedure

register allocator (in step-I). The *lscs* for a specific stacked register usage is the total spill cost of all live ranges using the stacked register in step-I. For simplicity, in this example, we assume that the cycles saved for each stacked register usage is the same for all stacked register usage in a specific procedure. We use *lscs* as the value of the load/store cycles saved by each stacked register per procedure invocation. Due to control flow in programs, a load/store operation is not always executed each time the procedure is invoked. Therefore the *lscs* value may be a fractional number.

We represent the caller-callee relations and the stacked register usage on a *Weighted Call Graph G*. Each edge (e.g. edge (p,q) where p is the caller and q is the callee) has the total invocation frequency of the procedure (e.g. number of times q is invoked by p) annotated on the edge. Each node representing a procedure is annotated with its stacked register usage estimation -- *sru* and *lscs*. **Figure 2(b)** shows the call graph G for example program in **Figure 2(a)**. Note that we have ignored the overlapping of input/output register frames between adjacent procedures in the call chain so far for simplicity, and an extension to take this into account is discussed in Section 6.

At step-II of the outlined, we want to identify "hotspots" – program points where the cost/benefit of stacked register allocation appears to be most significant. One way to identify a hotspot is to calculate the total accumulated stacked register usage, i.e. the product of edge frequency and the stacked register usage (*sru*) per invocation. For procedures A, B, C, D, E and F – the values are 60, 5000, 6000, 60, 5050, and 60 respectively. In this case, we identify C as the hotspot of the call graph, and it can grow to a hot region along the most frequently called path. Here the hot region can be viewed as a call-intensive trace. For example, procedure B is added to the hot region because that the invocation frequency from B to C is above a certain threshold. Similarly, A is also added to the hot region. However, D is not included because the call frequency of C→D is only 1, which is below the threshold. As a result, A→B→C is identified as one hot region. If we continue the process, D→E and F are identified as two other hot regions.

We can now perform stacked register quota assignment for each hot region. In order to reduce the load/store cycles for explicit spills as much as possible, stacked registers shall be assigned to procedures where the savings of spill cost exceed the cost of using the stacked registers. We can allocate the stacked register usage frame for the procedures in a hot region based on an order of their *lscs* values.

When the accumulated stacked register usage on the call path in a hot region exceeds 96, the quota assigned to some procedures during step-I shall be reduced in order to reduce the total spill-to-memory access cycles. In Section 3.1, we mentioned that measured cost for each register stack overflow/fill is about 1 cycle, and this cost is called *per_cost* in the rest of this paper. Hence, each single stacked register usage exceeding 96 will cost 100 cycles, since the invocation frequency of A→B→C is 100. In procedure B, each stacked register assignment could save 50 cycles (100x0.5 = 50) by eliminating the corresponding explicit load/store spill instructions, but it incurs 100 cycles of RSE cost. Hence we should not assign any stacked registers to B, and its quota will be taken away. This will save 2500 (50x50) load/store cycles. For procedure A, however, it is worth keeping its quota. At the end of processing, procedures A, B and C will be assigned a quota of 60, 0, and 60 stacked register usage, respectively. Similarly, we can perform the stacked register assignment for hot regions D→E and F. The final allocation is shown in **Figure 2(c)**.

3.3 Problem Formulation

For architectures, like Itanium® processors, which have a deep register stack, the spill-to-memory access time t_{mem_access} can be separated into two parts: memory access cycles associated to explicit load/store spill operations t_{spill_cost} and memory access cycles associated with RSE t_{rse_cost}. We often need to make a tradeoff between t_{spill_cost} and t_{rse_cost} for each procedure, i.e., between the explicit spills in intra-procedure register allocation in individual procedures and the inter-procedure register stack allocation, to minimize t_{mem_access}.

We assume that the following are given: weighted call graph *G* of a program P and hardware parameters in the target architecture, such as the upper bound of stacked registers (*bound*) and the cost for each register stack overflow/fill (*per_cost*), etc. We formulate the inter-procedure stacked register quota assignment problem as to determine stacked register usage budget for every procedure $p_i \in G$, which minimizes the overall spill-to-memory access time t_{mem_access}.

4 SOLUTION AND METHODOLOGY

We use a three-step process to implement the inter-procedural framework to allocate stacked registers for each

procedure.

I. Perform an estimation of intra-procedural register usage. In the current implementation, this estimation is obtained from the feedback of a first-pass intra-procedural register allocation.

II. Perform an inter-procedural stacked register quota assignment to guide the register allocation in step III based on the estimation from step I. The detailed algorithm of this step is in Section 4.1.

III. Use the result of step II as the input information to perform intra-procedural register allocation for each procedure.

4.1 Algorithm

In this section, a heuristic algorithm is proposed to perform the inter-procedural stacked register quota assignment (see step II above). The algorithm is shown in **Figure 3**. The inputs to the algorithm are: a weighted call graph G, the RSE cost per register stack overflow/fill (*per_cost*), and the upper bound of stacked registers (*bound*). Stacked register usage (*sru*) of every procedure and load/store cycles saved (*lscs*) by each stacked register usage are initialized based on the stacked register usage estimation of the intra-procedural allocation (step I).

The algorithm begins with function **Main(G)**. First, it takes a weighted call graph **G** and partitions G into a set of hot regions, **H**, by calling function **Find_Hot_Region(G)** (line 5)[1] repeatedly. For each given hot region **h** in **H**, every stacked register usage **r** used in a procedure **p** within **h** is inserted to a list **L** in a descending order of *lscs* by calling function **Construct_Sorted_List(h)** (line 9). Then quota assignment of stacked register usage is obtained by calling function **Quota_Assignment(L,h)** (line 10).

In the weighted call graph **G**, we are interested in only those hot regions where the cost/benefit of stacked register allocation appears the most significant. Function **Find_Hot_Region(G)** (line 12) is called repeatedly to partition G into hot regions. A node (procedure) with the maximum weight is selected from **G** as the initial hotspot. Here the weight of a procedure is the total accumulated stacked register usage, i.e., the product of its called frequency and stacked register usage. Because call-intensive procedures tend to have a high RSE cost, a hot region is extended forward from the hotspot to the most frequent callee if satisfying the following equation (line 21 to 26):

$$callee\_threshold(x,y) = \frac{call\_freq(x \to y)}{call\_freq(x)} > t \;\&\&\; \frac{called\_freq(y)}{called\_freq(hotspot)} > t'$$

Here *t* and *t'* are two threshold values decided by a compiler. The term *caller_threshold(x,y)* can be defined similarly, and the hot region can be extended backward from the hotspot. Function **Find_Hot_Region(G)** is called repeatedly to identify additional hot regions from the remaining procedures in G, i.e., the procedures not in any hot region yet.

The identified hot regions in set **H** are processed one at a time. Every stacked register usage **r** by the procedures in hot region **h** is inserted into a list **L** in a descending order of *lscs* by calling function **Construct_Sorted_List(h)** (line 35).

Each procedure **p** in a hot region is then assigned with a stacked register usage quota by calling function **Quota_Assignment(L,h)** (line 43). In this function, *count* is used to record the number of accumulated stacked register usage. It is initialized to zero (line 44). Stacked register usage **r** in **L** is checked one at a time (line 45). If the accumulated stacked register usage in the hot region does not exceed *bound* (line 47) and the procedure **p** in which **r** is used is not self-recursive (line 48), the usage is retained and the quota of procedure **p** is increased by one because there is no RSE cost incurred by its usage. If **p** is self-recursive, we assume that all stacked registers used in **p** will be overflowed to memory upon each invocation to itself. Therefore, if the accumulated stacked register usage does not exceed *bound* and **p** is self-recursive[2], the RSE cost caused by **r** is computed (line 52) as follows:

Call_Edge_Freq($p \to p$)**per_cost*

If *lscs* is greater than the RSE cost, the usage is retained and the quota of **p** is increased by one. At the same time, the value of *count* is also increased by one. Otherwise, this usage is eliminated to reduce the overall memory access time.

Here we also assume that if the accumulated stacked register usage exceeds *bound*, each additional stacked register usage in **p** causes a register stack overflow/fill per invocation of **p.** Hence, if the accumulated stacked register usage exceeds *bound*, the RSE cost caused by each additional stacked register usage could be computed as the product of the frequency **p** is called from its callers in the hot region and the RSE cost per register stack overflow/fill *per_cost* (line 59 to 60). Note that

[1] Here line 5 corresponds to line 5 in Figure 3.

[2] Our algorithm does not consider the mutual recursion because this is not common in our benchmarks, such as SPEC2000Int, and the extension to treat this problem is straightforward.

```
// Inputs: call graph G, sru (stacked register usage), and lscs (load/store cycles saved by register usage) for each
procedure, and register usage estimation of intra-procedural register allocation.
1    procedure Main(Call_Graph G) {
2        P = ∅; //P is the set of procedures already in hot regions

3        Hot_Region_Set H = ∅;      // the set of hot regions, initialized to empty.
4        do {
5            Hot_Region h = Find_Hot_Region(G); //Find a hotspot via traversing the call graph G
6            if (h) H = H ∪ h;
7        } while (h != ∅)

8        for each Hot_Region h∈ H {
9            List L = Construct_Sorted_List(h); // Construct a sorted list for hotspot h.
             // Assign each procedure a quota.
10           Quota_Assignment(L, h);
11       }}

     Hot_Region
12   procedure Find_Hot_Region(Call_Graph G) {
13       Hot_Spot h = ∅;
14        Call_Graph_Node hotspot = NULL;
15        max_weight = 0;
16        for each procedure p∈ G && p ∉ P {
17            if weight(p) > max_weight {
18                hot_spot = p; max_weight = weight(p); }
19        }
20        h = h ∪ {hotspot}; P = P ∪ {hotspot};

         //Extend from a hotspot to a hot region
21       Call_Graph_Node x = hotspot;
22       Call_Graph_Node y = the most frequently invoked callee in x;
23       while ( (y ∉ h) && callee_threshold(x,y) && y ∉ P){
24           h = h ∪ {y}; P = P ∪ {y};
25           x = y; y = the most frequent callee of x;
26       }

27       x = hotspot;
28       y = the most frequent caller of x;
29       while ((y ∉ hot_spot) && caller_threshold(x,y) && y ∉ P){
30           h = h ∪{y}; P = P ∪ {y};
31           x = y; y = the most frequent caller of x;
32       }
33       return h;
34   }

     List
35   procedure Construct_Sorted_List (Hot_Region h) {
36       L = ∅;
37       for each procedure p ∈ h {
38           for each stacked register r used in p {
39               insert r to L in a descending order of lscs(r);
40           }}
41       return L;
42   }

43    procedure Quota_Assignment (List L, Hot_Region h) {
44       count = 0; //Number of accumulated stacked register usage in hotspot h.
45       for every stacked register usage r in L {
46           Call_Graph_Node p = Procedure(r);// p is the procedure in which r is used
47           if (count < bound) {// the bound is 96 on Itanium
48               if (!Self_Recursive(p)) { // no RSE cost considered
49                   count++;
50                   p→quota++;
51               } else { // p is self-recursive
52                    cost = Call_Edge_Freq(p→p)*per_cost;
53                    if (lscs(r) > cost) { // Keeping r as a stacked register is beneficial
                          //The number of accumulated stacked register usage is increased by one.
54                        count++;
55                        p→quota++;//The quota of procedure p is increased by one.
56                    }
57               }//End of else
58           } else { //count >= bound, i.e.,accumulated stacked register usage exceeds bound
59               for each callee s of procedure p and s∈h
60                   cost = cost + Call_Edge_Freq(p→succ)*per_cost;
61               if (lscs(r) > cost) {
62                   count++;
63                   p→quota++;
64               }}}//End of else
65    }
```

Figure 3. Algorithm for Stacked Register Quota Assignment

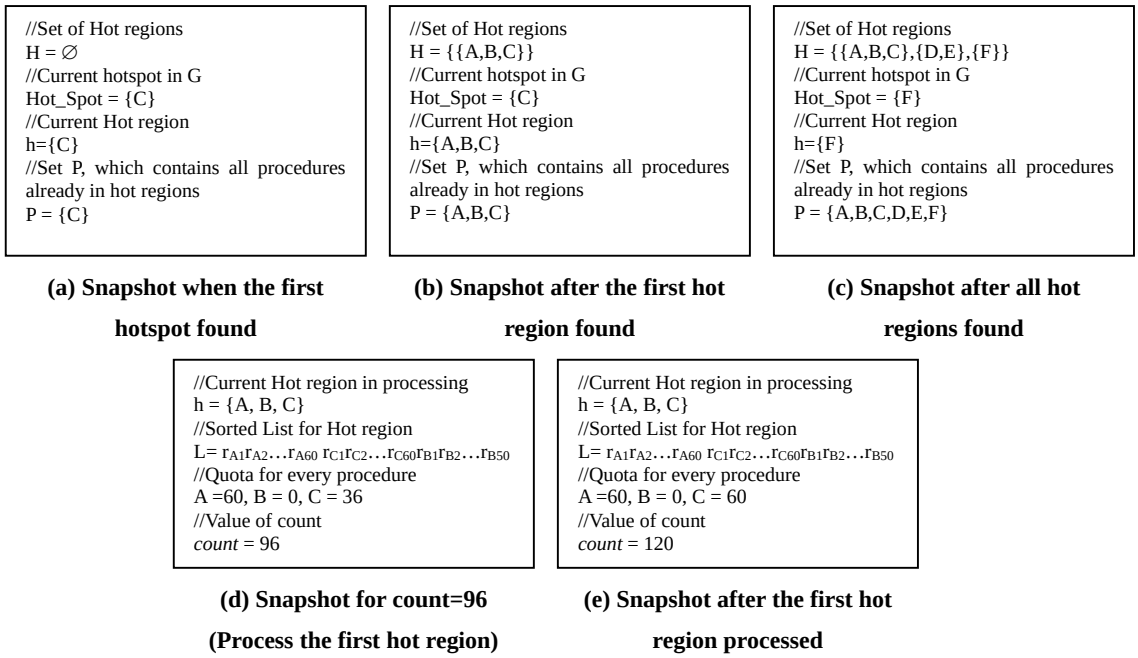

Figure 4. Snapshots of Application of Our Algorithm to the Motivating Example

the overlapping of input/output register frames between adjacent procedures in the call chain is ignored in the accumulated stacked register usage computation, and an extension to deal with this problem is discussed in Section 6. RSE cost and *lscs* are checked (line 61) to decide whether r should be kept or not. If a stacked register usage should be kept, the number of accumulated stacked register usage in the hot region is increased by one. Hence, every procedure gets a quota used to guide the intra-procedural register allocation in step III.

4.2 Time Complexity of Inter-procedural Quota Assignment Algorithm

Theorem 1 proves the time complexity of the algorithm (step II) is $O(n^2)$.

Theorem 1 *The time complexity of the algorithms in Figure 3 is $O(n^2)$, where n is the number of functions in the call graph.*

□ *Proof* : To traverse the call graph once is O(n). In the worst case, each time the traversed path will include only one function, and we need to traverse the call graph n times. The time complexity is $O(n^2)$.

Normally, each selected path includes more than one function, and after selecting the few top hot paths, all hot regions in the call graphs are formed. Hence, the average running time of the algorithm tends to be linear with respect to n in practice.

4.3 Case Study

Figure 4 shows a few snapshots in applying the algorithm to the motivating example in **Figure 2**. The procedure with the max weight is selected as the first hotspot. Here C has the maximum weight of 6000, so it is selected as the hotspot and placed in hot region **h** (line 20). The snapshot is shown in **Figure 4(a)**. The algorithm considers to extend the hot region from C to D, but D does not satisfy *callee_threshold(x,y)* (line 23). Therefore D is not included in the hot region. The algorithm also considers to extend the hot region backward from C to its caller B, and B satisfies *caller_threshold(x,y)* (line 29). Hence, B is added to the hot region (line 30). The hot region continues to extend backward from B to A as A satisfies *caller_threshold(x,y)* and is included in the hot region. **Figure 4(b)** shows the snapshot after the first hot region is identified. Among the procedures not yet in any hot region, the algorithm continues to identify more hot regions. The snapshot after all hot regions are found is shown in **Figure 4(c)**.

The three identified hot regions are processed one at a time (line 8 to 11). Hot region {A, B, C} is processed first. A sorted list **L** is constructed by calling **Construct_Sorted_List(h)** (line 9). L is shown in **Figure 4(d)**. The quota assignment for the current hot region is done by calling function **Quota_Assignment(L,h)**. When *count* equals to *bound* (the *bound* is 96 on Itanium®), the snapshot of processing the first hot region is shown in **Figure 4(d)**. Once *count* is greater than

96, each additional stacked register usage (from r_{A37} to r_{B50}) is decided by checking the *lscs* and the induced RSE cost (line 58 to 64). After the first hot region is processed, the quota for A, B, and C are 60, 0, and 60 respectively, as shown in **Figure 4 (e)**. The other two hot regions are then processed, and the final result is shown as **Figure 2(c)**.

5 EXPERIMENTAL RESULTS AND ANALYSIS

5.1 Experimental Environment

The hardware platform for our experiments is based on an HP machine with a 733MHz Itanium® processor and 2 Mbytes L2 cache. The implementation is done in the Open Research Compiler (ORC) [11]. ORC employs many Itanium® specific optimizations, such as if-conversion, data and control speculation, instruction scheduling integrated with micro-scheduling, and predicate analysis, etc. Its global register allocation is an integration of register allocation approaches by Chow [2][8] and Briggs [3], respectively. SPEC2000Int programs are used to evaluate the effectiveness of our approach. All SPEC2000Int binaries are generated by ORC with all of the optimizations, which include intra-procedural optimizations, profiling feedback, inter-procedural analysis, and function inlining.

5.2 Experimental Results

Our experiments show the following main results:

1 Performance of execution is improved significantly for some programs. Our experiments show this algorithm is quite effective for some benchmarks, raising the execution performance up to 14%.

2 For other benchmarks without performance improvements, our algorithm has no negative effect to the execution performance except for **gzip** with slight performance degradation of 1.2%.

3 Through further analysis, we find that all those benchmarks with obvious performance improvement have considerable RSE cost. Because our algorithm reduced the RSE cost of these benchmarks effectively with the register spill cost under control, the total effect is positive in cutting the overall memory access time.

4 An analysis shows that the stacked register usage in some call-intensive procedures is clearly reduced, which leads to significant RSE cost reduction in benchmarks.

5 Our algorithm appears to be effective in finding the hot regions causing the most RSE cost in the weighted call graph of these benchmarks.

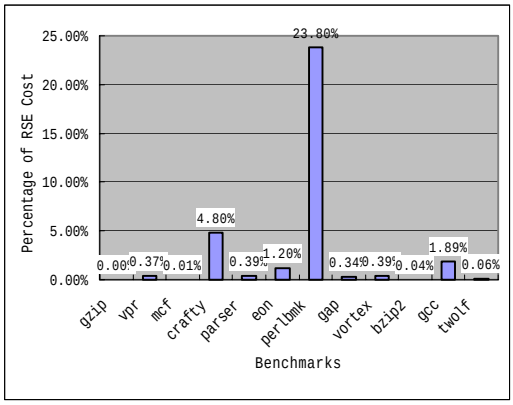

Figure 5. RSE Percentage of Total Execution Time

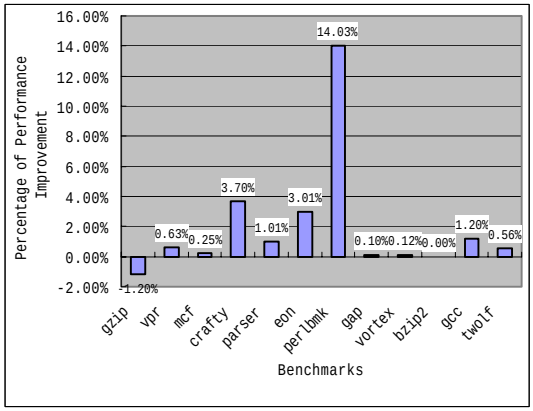

Figure 6. Performance Improvements due to Inter-procedural Quota Assignment of Stacked Registers

Figure 5 shows the percentage of RSE cost in the total execution time for all 12 SPEC2000Int programs. Program **perlbmk** has almost 24% of execution time on RSE cost, and **crafty** has 4.8% on RSE cost. The RSE issue is significant in these two programs. Programs **gcc** and **eon** also have non-trivial RSE costs, which account for 1.89% and 1.2%, respectively. The RSE cost is insignificant in the other programs, below 0.5%.

Figure 6 shows the percentage of performance improved by the proposed algorithm. All test cases with noticeable RSE costs have clear performance improvements. Program **perlbmk** has the highest improvement at 14%. For **crafty**, **eon** and **gcc**, the performances are improved by 3.7%, 3.01%, and 1.2%, respectively. Another observation is that the programs with negligible RSE costs do not have adverse performance impact,

except for gzip degraded slightly by 1.2%.

5.3 Analysis

In order to optimize between RSE cost and spill cost due to the allocation of stacked register usage, the algorithm proposed in this paper must achieve the following two objectives:

(A) In the call graph, the algorithm can identify the hot regions responsible for the most RSE cost in the program.

(B) It can make a good tradeoff between RSE cost and spill cost introduced by reducing stacked register usage in call-intensive functions.

In SPEC2000Int, **perlbmk**, **crafty** and **gcc** are chosen to be analyzed to demonstrate the effectiveness of the proposed approach. **Perlbmk** and **crafty** are chosen because of their performance improvement. **Gcc** is chosen because it is a large and complex program.

We will show (*A)* first whether an algorithm could find out hot regions responsible for the most RSE cost. We will then show if the additional stacked register usages exceeding *bound* in these hot regions are greatly reduced, most of the RSE cost should be eliminated. In our algorithm, any additional stacked register usage exceeding *bound* will not be permitted if the incurred RSE cost is greater than the cycles saved by its usage. Since RSE cost is *per_cost*called_freq(p)*, we could set the *per_cost* extremly high to reduce most RSE cost. Here we choose the top 10 hottest regions from the weighted call graph of each program and set the *per_cost*(RSE cost per register stack overflow/fill) to be 20 cycles, which is much higher than the real cost (about 1 cycle, see Section 3.1), to reduce most of the RSE cost without considering the increase of register spill cost.

Experimental results in **Figure 7** show that the RSE cost in **perlbmk** and **crafty** almost disappeared (the bar represents RSE cost negligible after optimization in **perlbmk**), while the RSE cost in **gcc** is also reduced from 1.5% to 0.5%. The experiments show that this algorithm appears to be effective in finding hot regions causing most of the RSE cost.

Next we will show (B) that this algorithm makes a good tradeoff between the intra-procedure spill cost and the inter-procedural RSE cost. In **Figure 8** we can see that when the algorithm is employed, **crafty**, **perlbmk,** and **gcc** have their total spill-to-memory access time reduced to obtain 3.2%, 18%, and 0.9% reduction in the total execution time, respectively. Performance improvement for **perlbmk** is 13%, which is less than the spill-to-memory access time reduction in the overall execution time. On the other hand, **gcc** improves its total performance higher than the gain in the spill-to-memory access. This is due to different results from instruction scheduling, which improves the quality of scheduling**.**

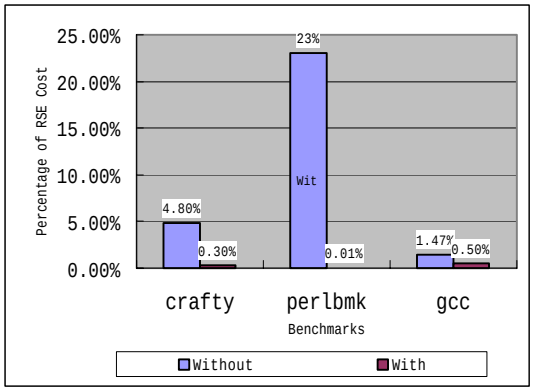

Figure 7. Percentage of RSE Cost With and Without Reducing Stacked Register Usage Greatly in Hot regions

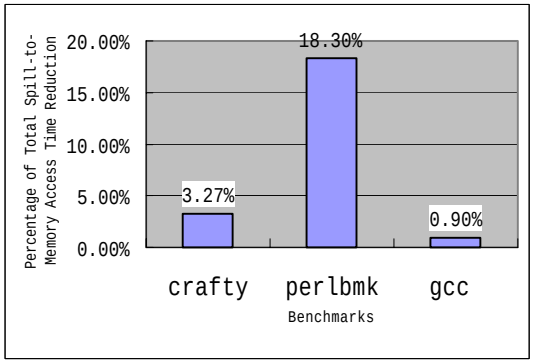

Figure 8. Percentage of the Reduction in Spill-to-Memory Access Time over Total Execution Time

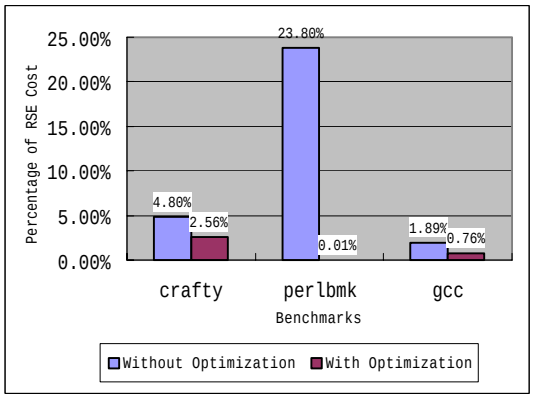

Figure 9. Percentage Comparison of RSE Cost in Overall Execution Time With and Without Optimization

Figure 9 shows the percentage of the RSE cost in overall execution time with and without optimizations. According to **Figure 9** almost all of the RSE cost disappears in **perlbmk** after the optimization. For **crafty** and **gcc**, though the RSE cost with optimization is clearly reduced compared to that without optimization, but some RSE cost still remains. From the above discussion, we know that the RSE cost reduction is at the expense of register spill cost. The key to this algorithm is to make a good tradeoff between them.

Table 1 Register Spill Cost Comparison between With and Without RSE Optimization

Function Name	Crafty	Perlbmk	Gcc
Spill Cycles without RSE Optimization	3.96E+06	2.5E+03	1.06E+07
Spill Cycles with RSE Optimization	9.74E+07	2.6E+09	2.25e+07

Table 2 Stacked register Usage Comparison between With and Without RSE Optimization

Function Name	Perlbmk Regmatch	Crafty Evaluate	Crafty EvalutePawns
Without	73	35	52
With	25	27	21
Function Name	Gcc Recog_5	Gcc Canon_reg	Perlbmk Perl_pp_method
Without	13	21	13
With	10	16	11

We also count the number of cycles due to register spills. **Table 1** shows that all three programs have register spill cost increased after RSE optimization. **Perlbmk** has an increase of 2.6E+09 cycles in register spills, which explains why while almost all of the RSE cost (24%, **Figure 9**) is eliminated, the overall spill-to-memory access time reduction is only 18% (**Figure 8**). From these data, we know that though the register spill cost increases, it is compensated by a larger reduction in RSE cost. Therefore, this leads to the reduction of total spill-to-memory access time in **Figure 8**.

Table 2 compares the stacked register usage of some call-intensive procedures. We can see that these procedures use fewer stacked registers after RSE optimizations. For example, the *regmatch* procedure in **perlbmk** uses 73 stacked registers during intra-procedure register allocation, but uses only 25 stacked registers by adopting the proposed approach. The above data show that this algorithm can make a good tradeoff between RSE cost and register spill cost by reducing the stacked register usage in procedures of hot regions guided by the assigned quota.

6 DISCUSSION

There are two aspects in which we can further refine our algorithm: the overlaps of register stack frames and region selection. As computing the accumulated stacked registers within a region, the current implementation of our approach double counts the overlap areas that are used for passing arguments between two adjacent procedures. Double counting the register usage of argument passing may cause an over estimation of the number of accumulated stacked, resulting in reducing quotas allocated to procedures.

The current region selection is essentially a call-invocation trace. The trace is expended toward both directions by traversing the most frequently invoked callee/caller path—from a node to its most frequently invoked callee/caller, respectively. Under some circumstances, the call-invocation trace may not capture a global view of register usage. For example, for the contrived call graph depicted in **Figure 10**, our algorithm selects C as the seed and extends the hot region along the callers to get a hot region R that is A→B→C.

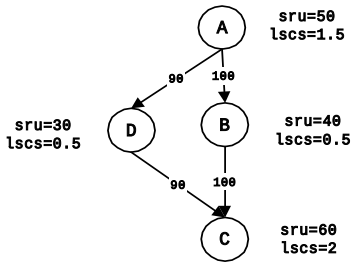

Figure 10. Example of Call Graph

Although the call edge frequency C→D is also high, D is not on the most frequently invoked call path and therefore is not included by the hot region. When selecting the next hot region, our algorithm excludes A, B and C because they have already been in region R. Hence, the second hot region is D itself. The current quota assignment assigns A, B, and C with register quotas 50, 0 and 50, respectively. Procedure D's quota is 30. The quota assignment incurs high RSE cost along the path A→D→C. To alleviate the issue, we can refine the region selection to form a region that contains multiple call paths instead of just one single path. The register allocation then tries to balance stacked register assignment among multiple call

paths so as to reduce the RSE cost from the whole region's perspective.

In this algorithm, we assumed that if the accumulated stacked register usage within hot region exceeds *bound*, each additional stacked register usage will cause a register stack overflow/fill. Though this assumption is correct for most conditions, it is not always true.

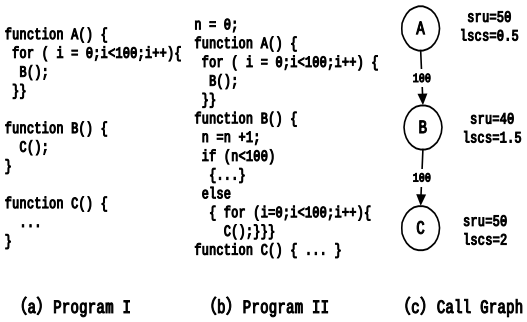

Figure 11. Programs with Same Call Graph

In **Figure 11**, there are two programs(I and II) having the same weighted call graph G. For program I, if the accumulated stacked register usage along a call path in G exceeds *bound*, each additional stacked register usage will cost 100 cycles; but for program II, each additional stacked register usage of C exceeding *bound* only cause 1 cycle, for the first time C is called by B. For the other 99 times invocation of C, because register stack contains only register stack frames of B and C, and the accumulated stacked register usage of the two procedures does not exceed *bound*, so there is no RSE cost. One can do value instrumentation of dynamic average stack fill/spill cost for every procedure to get more accurate feedback. We are improving our algorithm in this area as possible future work..

7 RELATED WORK

Douillet et al. [12] proposed a multi-alloc method in order to reduce overflows and fills from backing store. Because different control flow paths use different numbers of stacked registers, by inserting multiple *alloc* instructions on different paths, the stacked register usage could be reduced in some paths. Although the method is interesting, their experimental results are not satisfactory. Weldon et al. [13] did a quantitative evaluation of the register stack engine and proposed a hardware-based optimization algorithm to eliminate unnecessary spill/restore on different control flow paths. In ORC, stacked registers are classified into two types: stacked callee registers and stacked caller registers. For those live ranges that need stacked registers but do not cross call sites, the compiler assigns them stacked caller registers, i.e, *output* register. With this method, the overlapping part of input/output areas between two adjacent procedures is enlarged. As such, more register stack frames are likely to fit within 96 physical stacked registers. Settle et al. [14] proposed a method of inserting multiple *alloc* instructions at call sites depending on liveness analysis. But the effectiveness of their method was not clear from their experimental results.

8 CONCLUSION

In Itanium® architecture, without modeling the cost of stacked registers, the compiler may hold a perception that using more stacked registers to eliminate spill cost is always a win. In reality, sometimes, the cost of using stacked registers exceeds the saving of spills. Our experimental results indicate that this is likely to cause high RSE overhead for the programs that are call intensive and have deep call chains. In this paper, we propose a new algorithm that reduces RSE cost based on an inter-procedural register allocation framework. Our approach uses a quantitative cost model of stacked register usage. The inter-procedural framework assigns a register quota for each procedure based on a tradeoff analysis between spill saving and RSE overhead. The results show that our approach is effective and improves performance significantly for the programs with high RSE overflow cost, e.g. **perlbmk** and **crafty**, improved by 14% and 3.7%, respectively. Our approach does not appear to cause performance deterioration to other programs without much RSE cost.

ACKNOWLEDGEMENTS

We would like to acknowledge all members contribute to the ORC project. We would also thank Fred Chow for his helpful discussions and insights, Robert Cox and the anonymous referees for their useful comments. This work is supported by Intel Company, Chinese Science and Technology Ministry (Grant No.2001AA111061 and Grant No.2002AA1Z2104).

REFERENCE

[1]Chaitin, G Register allocation and spilling via graph coloring. In Proceedings of the SIGPLAN 82 Symposium on Compiler Construction (Boston, Mass., June 1982). ACM, New York, 1982, pp. 98-105.

[2] Chow, F. C., and Hennessy, J. L.. Register allocation by priority-based coloring. In Proceedings of the SIGPLAN 84 Symposium on Compiler Construction (Montreal, June 1984). ACM, NewYork, 1984, pp. 222-232.

[3] P. Briggs, Register Allocation via Graph Coloring. PhD thesis, Rice University

[4] P.Briggs, K. Cooper, and L. Torczon. Improvements to graph coloring register allocation. ACM Transactions on Programming Languages and Systems, 16(3):428-- 455, May 1994.

[5] G. Lueh and T. Gross. Call-cost directed register allocation. In Proc. ACM SIGPLAN '97 Conf. on Prog. Language Design and Implementation, pages 296--307. ACM, June 1997. ACM Transactions on Programming Languages and Systems.

[6] Peter A Steenkiste and John L. Henessy,A simple interprocedural register allocation algorithm and its effectiveness for LISP.Transactions on Programming Languages and Systems,pages 1-30,January 1989.

[7] David W. Wall. Global Register Allocation at Link Time. In Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, pages 264-275, New York, 1986

[8] Fred C. Chow,Minimizing Register Usage Penalty at Procedure Calls,In Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation,June 1988.

[9] S. M. Kurlander and C. N. Fisher, “Minimum Cost Interprocedural Register Allocation”, Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1996, pp.230-241, 1996.

[10] Intel Corporation. Intel IA-64 Architecture Software Developer’s Manual. Santa Clara, CA, 2000.

[11] Open Research Compiler for Itanium Processors, http://ipf-orc.sourceforge.net, Jan 2003.

[12] A. Douillet, J. N. Amaral, and G. R. Gao. Fine-grained stacked register allocation for the Itanium architecture. In 15th Work-shop on Languages and Compilers for Parallel Computing(LCPC), 2002.

[13] R. David. Weldon, Steven S. Chang, Hong Wang, Gerolf Hoflehner, Perry H. Wang, Dan Lavery, and John P. Shen, Quantitative Evaluation of the Register Stack Engine and Optimization for Future Itanium Processor,In proceedings of the 6th Annual Workshop on Interaction between Compilers and Computer Architectures,Boston, 2002.

[14] Alex Settle and Daniel A. Connors, Optimization for the Intel Architecture Register Stack, International Conference of in Proceedings of International Symposium of Code Generation and Optimization (CGO), San Francisco, March, 2002.

.