

Optimal Register Reassignment for Register Stack Overflow Minimization

YOONSEO CHOI and HWANSOO HAN
Korea Advanced Institute of Science and Technology

Architectures with a register stack can implement efficient calling conventions. Using the overlapping of callers' and callees' registers, callers are able to pass parameters to callees without a memory stack. The most recent instance of a register stack can be found in the Intel Itanium architecture. A hardware component called the register stack engine (RSE) provides an illusion of an infinite-length register stack using a memory-backed process to handle overflow and underflow for a physically limited number of registers. Despite such hardware support, some applications suffer from the overhead required to handle register stack overflow and underflow. The memory latency associated with the overflow and underflow of a register stack can be reduced by generating multiple register allocation instructions within a procedure [Settle et al. 2003]. Live analysis is utilized to find a set of registers that are not required to keep their values across procedure boundaries. However, among those dead registers, only the registers that are consecutively located in a certain part of the register stack frame can be removed. We propose a compiler-supported register reassignment technique that reduces RSE overflow/underflow further. By reassigning registers based on live analysis, our technique forces as many dead registers to be removed as possible. We define the problem of optimal register reassignment, which minimizes interprocedural register stack heights considering multiple call sites within a procedure. We present how this problem is related to a *path-finding problem* in a graph called a *sequence graph*. We also propose an efficient heuristic algorithm for the problem. Finally, we present the measurement of effects of the proposed techniques on SPEC CINT2000 benchmark suite and the analysis of the results. The result shows that our approach reduces the RSE cycles by 6.4% and total cpu cycles by 1.7% on average.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*compiler; code generation; optimization*; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream processors*; C.5.3 [Computer System Implementation]: Microcomputers

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Register assignment, register allocation, register stack, sequence graph

Authors' address: Yoonseo Choi and Hwansoo Han, Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 373-1 Guseong-Dong, Yuseong-Gu, Daejeon 305-701, Korea; email: yschoi@arcs.kaist.ac.kr, hshan@cs.kaist.ac.kr

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2006 ACM 1544-3566/06/0300-0090 \$5.00

1. INTRODUCTION

Memory latency frequently limits the performance of applications, despite the advances and innovations in modern architectures. Smart register allocators can play an important role in reducing the number of register spills/fills, which are intrinsically load/store operations. Typical register allocators focus on the reduction of register spills/fills within a procedure boundary. Another important issue in register allocation, however, exists regarding procedure boundaries. When a new procedure is invoked, all the live local registers used in current procedure must be saved in memory. When the procedure returns to the caller, the caller can resume its execution after restoring the register context from the memory. Many researchers investigated register allocation techniques to reduce the memory access penalty associated with procedure calls [Wall 1986; Chow 1988; Steenkiste and Henessy 1989; Kurlander and Fisher 1996; Lueh and Gross 1997].

Microarchitectural solutions to this problem can be found in many commercial microprocessors. Providing large physical register files and maintaining them in register windows [Weaver and Germond 1994] or register stack [Intel Corporation 2002] can mitigate the overhead of register saves/restores around procedure boundaries. The SPARC architecture allocates a register window per procedure which consists of 24 registers [Weaver and Germond 1994]. First, eight registers are used for procedure arguments (*in-registers*). The next eight registers are used for local registers for the current procedure (*local-registers*). The last eight registers are used to pass arguments to callee procedures (*out-registers*). Caller's *out-registers* are overlapped with callee's *in-registers*. The Itanium architectures provide a more flexible mechanism called a register stack, using a special instruction (*alloc*) that can control the sizes of *in/local/out-registers* for each procedure [Intel Corporation 2002].

Both the register stack and register window offer a fast argument-passing mechanism among procedures. More importantly, they can reduce the overhead of register saves/restores on procedure calls that typically otherwise require memory accesses. Hardware-managed register stacks and register windows are key components, which provide the illusion of the infinite length of registers by automatically backing up and reloading the contents of physical registers from the memory without the explicit intervention of software. One of the current trends in microprocessors is to equip them with large physical registers though the number of visible registers from applications is fixed. Both the register stack and register window can benefit from large physical register files by having to back up and reload the contents of registers less frequently. Dynamically resizing the register stack frame can further circumvent a significant amount of register spill/fill overhead involved in the register stack overflow and underflow. Settle et al. [2003] proposed a *multi-alloc* technique, which generates multiple register allocation instructions to reduce the register stack height of all procedures in the call stack. Since there is a limited number of physical registers, for example, 96 in the case of Itanium, those reductions in the register stack height can subsequently lead to reductions in register stack overflow and underflow. The key idea is that nonlive registers across calls overlap with the callee's register stack frame. Register liveness analysis is used

to distinguish the registers that can be overlapped. Although Settle et al. provided a great deal of useful evidence that an overlapping register stack frame would lead to register stack engine (RSE) overhead reduction, their method takes advantage of only the nonlive registers at the end of the local-register region.

We found that carefully distributing registers to proper parts of the local-register region helps to reduce the register stack height by enlarging the number of the overlapped dead registers. Our method also generates multiple register allocation instructions within a procedure. However, we devised a compiler-supported register reassignment scheme to maximize the overlap with the callee's register stack frame. In particular, our scheme considers multiple call sites within a procedure for the overall minimization of the register stack height. Our contributions are as follows.

1. We define the problem of finding the optimal register reassignment that minimizes the register stack height of all procedures in the call stack, considering multiple call sites within a procedure.
2. We formulate the optimal register reassignment problem as the path-finding problem in a graph called *sequence graph* and show the equivalency of the two problems.
3. We also propose a heuristic algorithm to solve the problem.

The remainder of this paper is organized as follows. Section 2 presents background information on the register stack. Section 3 explains how we exploit register reassignment to minimize the register stack height interprocedurally. The section goes on to define the problem of optimal register reassignment and describe how the problem is related to *sequence graphs*. Section 4 presents our register reassignment algorithms based on path-finding on a sequence graph. Finally, Section 5 reports on the experimental results and analysis.

2. BACKGROUND

The Itanium architectures provide 128 general-purpose registers ($r0-r127$), which are virtual registers visible to applications. First 32 registers ($r0$ to $r31$) are global registers visible to all procedures; the remaining 96 registers ($r32-r127$) are stacked registers that are locally accessible within a procedure. Among the 96 stacked registers, a procedure claims a portion of them for use within the procedure. The actual number of physical registers for a register stack is usually larger than the number of virtual stacked registers. A register stack consists of those physical registers and the register stack engine (RSE). The stacked registers claimed by a procedure are mapped on to physical registers; we call them the register stack frame for that procedure.

Each register stack frame is grouped into three parts: (1) *in-registers* for incoming parameters from callers, (2) *local-registers* for the current procedure's use during its lifetime, and (3) *out-registers* for output parameters passed to callees. *Rotating-registers*, which are used for software-pipelined loops, are optionally selected among *in-registers* and *local-registers*. The size of the register stack frame is often determined at the beginning of procedure by invoking a

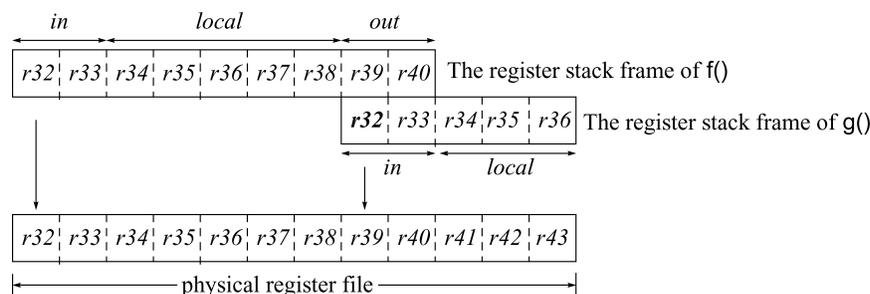


Fig. 1. Mapping a register stack frame to a physical register file, when $f()$ calls $g()$.

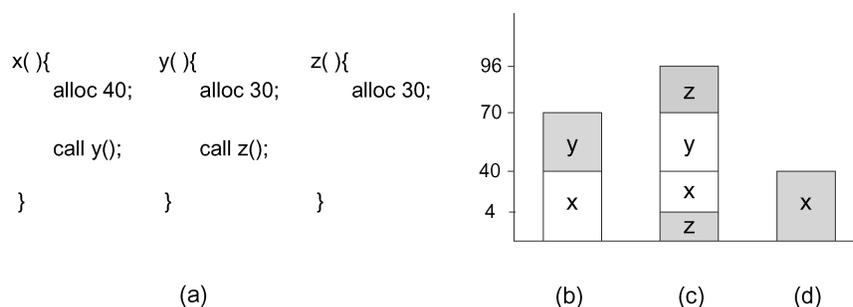


Fig. 2. The overflow and underflow of registers by RSE when x , y , and z are called in order: (a) example procedures x , y , and z ; (b) after y is called; (c) after z is called, four of the x 's registers overflow; (d) after y returns, four of the x 's registers underflow.

special instruction, *alloc*. The format of the instruction is

$$alloc \langle targ_reg \rangle = ar.pfs, in, local, out, rot$$

where *ar.pfs* is a special register to save the previous state [Intel Corporation 2002]. The total number of registers in the register stack frame for a procedure is $\#in\text{-registers} + \#local\text{-registers} + \#out\text{-registers} \leq 96$, with conditions of $\#in\text{-register} \leq 8$ and $\#out\text{-registers} \leq 8$.

To provide uniform accesses to its stacked register frame, the starting register of each register stack frame is remapped to $r32$ and the following registers are also remapped in order. The mapping between virtual registers and physical registers is illustrated in Figure 1, where procedure f calls g . The parameters are passed from f to g through the overlapping of the out-registers of f and the in-registers of g . The register stack engine (RSE) manages the mapping of register stack frame to physical registers in a circular manner [Intel Corporation 2002]. If multiple register stack frames use up all the physical registers, RSE overflows some of the stacked registers to a backing store in memory. When returning from a procedure, the previous register stack frame must be restored. If there are any overflowed registers, the RSE underflows the requested registers back to the physical registers from the backing store. Figure 2 shows how RSE overflows and underflows registers as procedures are called and return. (The overlapping of the caller's register stack frame with the callee's is ignored for ease of description. We assume the size of a physical register file is 96.) In

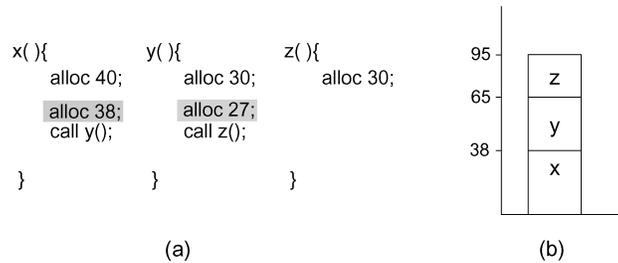


Fig. 3. Avoidance of the overflow/underflow of registers by utilizing multiple *alloc* instructions.

Figure 2a, procedure x calls y , y calls z . Procedures x , y , and z allocate 40, 30, and 30 stacked registers, respectively. Suppose that the register stack is configured, as shown in Figure 2b, when y is called by x . The register stack frame of the current active procedure is denoted with a gray box. When z is called by y , four of x 's registers overflow and are stored in memory by RSE to keep z 's registers in the physical register file. When z and y return subsequently, the four registers of x are restored from the backing store for further execution of x .

Now we demonstrate how we can avoid the overflows and underflows of registers by generating multiple register allocation instructions. If we insert additional *alloc* instructions before x calls y and y calls z , as in Figure 3a, none of the x 's registers overflow when z is called, because the total of registers allocated by x , y , and z does not exceed 96 (refer to Figure 3b). By carefully analyzing the liveness of registers, additional *alloc* instructions can reduce the total height of the register stack frames.

3. THE OVERFLOW-MINIMIZING REGISTER REASSIGNMENT

To determine the size of the register stack frame, procedures often start with *alloc* instructions. Suppose procedure f allocates nine stacked registers, $r32$ through $r40$, (in-registers: $r32, r33$, local-registers: $r34$ to $r38$, out-registers: $r39, r40$), as shown in Figure 4a and b. When f calls g , the out-registers of f overlap the in-registers of g , as shown in Figure 4b. Figure 4c illustrates multi-*alloc* method in [Settle et al. 2003]. Equipped with careful live register analysis, the register stack frame of f can be reduced by issuing another *alloc* instruction before calling g , which removes two dead local-registers, $r37$ and $r38$, and reassigns two output registers, $r39$ and $r40$, to $r37$ and $r38$, respectively. This resizing of f 's register stack frame decreases the register stack height for f and g by 2.

Now suppose f calls another procedure h besides g , as shown in Figure 4d. We can still add *alloc* instruction before calling g . However, before calling h , we cannot reduce the register stack frame of f in spite of dead local-register $r37$. If we reassign some of the local-registers of f , for example, by interchanging $r35$ and $r36$ and interchanging $r37$ and $r38$, we can reduce f 's register stack frame before calling both g and h , which leads to the reduction of the overall register stack height by 3 and 1 (refer to Figure 4f). This example clearly reveals the effect of register reassignment on the overall register stack height

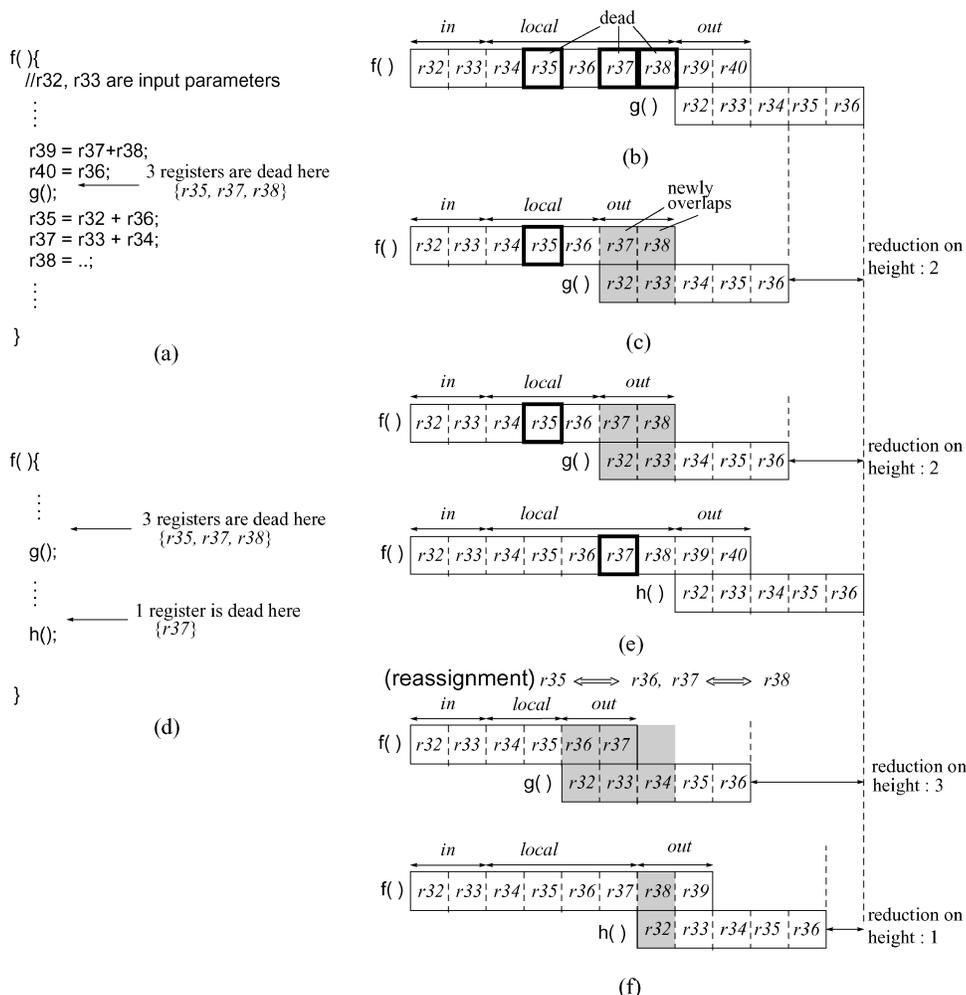


Fig. 4. Register stack height reduction: the resizing of the register stack frame and the effect of register reassignment.

reduction when there is more than one call site in a procedure. We observe that, by scanning from the last local-register (i.e. the local-register that has the biggest number among all local-registers) to the first local-register, we can eliminate the registers one by one from the current register stack frame while these registers are dead. Once we encounter a live local-register, we stop the scanning, even though there are more dead local-registers beyond it. We can only remove those scanned dead local-registers that are consecutively located at the end region of local-registers. Thus, pushing dead registers toward the end of the local-register region is a key concept to further reduce current register stack frame.

According to the calling convention, *out-registers* naturally overlap with the callee’s register stack frame and *in-registers* convey parameters passed by

callers of current procedures. The only candidates for safe reassignment are *local-registers*. For these local-registers, all virtual register names are mutually interchangeable. However, when a procedure uses rotating-registers for software pipelining, not all local-registers are interchangeable. We excluded the procedures that use rotating-registers from our optimization. Once we find one-to-one interchange mapping among local-registers, we can safely reassign local-registers that result in more overlaps. The rest of this section discusses (1) how we define optimal register reassignment, and (2) how we convert the register reassignment problem to a graph problem.

3.1 Optimal Register Reassignment

For a procedure P , if there are k call sites C_1, C_2, \dots, C_k then the set of registers which are dead before the call site C_i (or unused until that point) is denoted as dead set D_i . Dead set D_i has weight $w(D_i)$ according to the relative importance of the call site C_i . Let $L = \{r_1, r_2, \dots, r_l\}$ be the set of *local-registers* of the register stack frame for P . We now want to find a sequence $S, s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_l, s_i \in L, 1 \leq i \leq l$, of all stacked registers in L , which maximizes \mathcal{G} :

$$\mathcal{G} = \sum_{i=1}^k \text{gain}(S, D_i) = \sum_{i=1}^k \text{gain}(s_l, D_i) , \quad (1)$$

$$\text{gain}(s_j, D_i) = \begin{cases} w(D_i) + \text{gain}(s_{j-1}, D_i) & \text{if } s_j \in D_i, j = l \\ w(D_i) + \text{gain}(s_{j-1}, D_i) & \text{if } s_{j+1} \in D_i \text{ and } s_j \in D_i, 1 \leq j \leq l - 1 \\ 0 & \text{otherwise} \end{cases}$$

When the original sequence of *local-registers* S_0 is $r_1 \rightarrow r_2 \rightarrow \dots \rightarrow r_l$, we map the i th register in S_0 to the i th register in the new sequence $S, (1 \leq i \leq l)$; then we find this bijective function $RA : L \rightarrow L$ as our register reassignment. The gain of the register reassignment RA is \mathcal{G} given in Eq. (1).

Equation (1) represents the observation that the register stack frame can be overlapped from the right end until it encounters the first live register in the sequence. The reason why we can overlap only the right side of the sequence is that register frames for procedures are maintained in the form of a stack. We can stack up a new register frame on top of the current frame, which means a new register frame can be overlapped only with the right end of the current register sequence. The meaning of the $\text{gain}(S, D_i)$ is the number of registers that can be overlapped at call site C_i . The total gain, \mathcal{G} , is a plain sum of all the gains at all call sites in the procedure P .

For example, suppose a procedure has three call sites and their dead sets are $D_1 = \{r45, r46, r48\}$, $D_2 = \{r43, r45, r48\}$, and $D_3 = \{r43, r45\}$, where $w(D_1) = w(D_2) = w(D_3) = 1$ and the set of all stacked registers in local region $L = \{r42, r43, \dots, r48\}$. The gain of a sequence, $S = r42 \rightarrow r44 \rightarrow r46 \rightarrow r47 \rightarrow r43 \rightarrow r45 \rightarrow r48$, is then computed to be 5. For the first call site, we can overlap only the last two registers ($\text{gain}(S, D_1) = 2$). For the second call site, we can overlap the last three registers from the sequence ($\text{gain}(S, D_2) = 3$). Finally, for the third call site, none of the registers in the sequence is overlapped ($\text{gain}(S, D_3) = 0$). One thing to note is that only the suffix of S , which consists

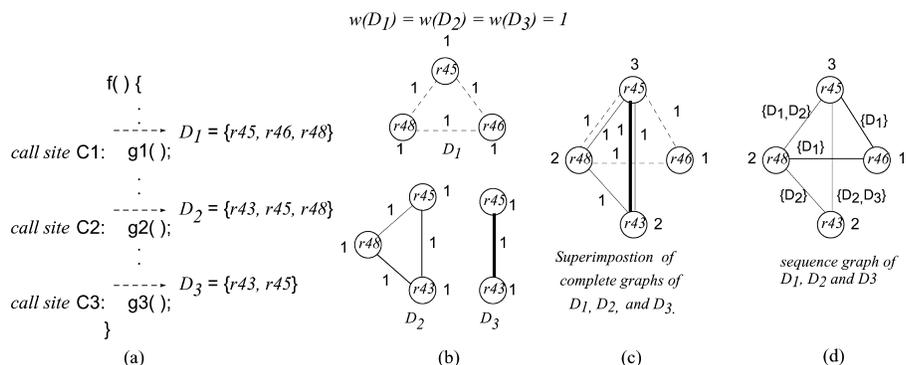


Fig. 5. An example of a sequence graph.

of dead registers matters. The order of the remaining registers is irrelevant to the total gain.

The optimal register reassignment is the register assignment that maximizes the gain in Eq. (1).

3.2 Optimal Register Reassignment Using Sequence Graphs

Our register reassignment is more easily tackled by relating it to a graph optimization problem, as shown in Figure 5. Converting the original problem into a graph problem gives us more insight on how to solve our problem. To find a register reassignment that produces maximum gain, we attempted to maximize the number of overlapped registers not just in one dead set, but in all dead sets. Since we can overlap with dead registers from the right end of a sequence, we tried all possible permutations of dead registers in all dead sets of all call sites. That means we attempted to find a permutation that results in the maximum total gain.

Now, using an example, we explain how we convert our problem to an equivalent graph problem. Figure 5a shows three dead sets D_1 , D_2 , and D_3 from our previous example, where a function f calls $g1$, $g2$, and $g3$ at three call sites C_1 , C_2 , and C_3 , respectively. $w(D_1) = w(D_2) = w(D_3) = 1$. In Figure 5b, complete graphs for dead sets D_1 , D_2 , and D_3 are shown. Any possible sequence of the registers of each dead set can be represented as a path on each complete graph. The weights of nodes and edges are 1 according to the weight of the corresponding dead sets. The superimposition of the three complete graphs is shown in Figure 5c, where the edges from complete graph D_1 are denoted by dotted lines, from D_2 by solid lines, and from D_3 by a heavy line. When we compose a superimposed graph, the weight of each node is the sum of the weights from all original graphs.

Recall that, in the previous example, we picked the sequence $S = r42 \rightarrow r44 \rightarrow r46 \rightarrow r47 \rightarrow r43 \rightarrow r45 \rightarrow r48$ ending with $r43 \rightarrow r45 \rightarrow r48$. In D_1 , only $r45$ and $r48$ can be overlapped from the local part of the register stack frame, since $r46$ does not appear consecutively with the other dead registers, $r45$ and $r48$, in the sequence. Whereas, $r43$, $r45$, and $r48$ in D_2 can all be overlapped since all of them appear consecutively at the end of the sequence.

In D_3 , no register is overlapped, and, as a result, we could reduce register stack frame by 2 and 3 before call site C_1 and C_2 , respectively.

Turning to our graph notations, on the complete graph of D_1 , we select the path $r48 \rightarrow r45$. On the complete graph of D_2 , we select the path $r48 \rightarrow r45 \rightarrow r43$. Note the orders of the registers in the paths: they are in the reverse order of sequence S . Projecting the paths to the superimposed graph in Figure 5c and adding the weights of the first node $r48$ and the following edges, $(r48, r45)$ and $(r45, r43)$, we obtain $5 = 2 + (1 + 1) + 1$, which is the same as the total gain (in Eq. 1) of the sequence S mentioned above. Note that for edge $(r45, r43)$, we added only 1, the weight of D_2 , excluding that of D_3 . Dead set D_3 does not have $r48$, the first node of the path $r48 \rightarrow r45 \rightarrow r43$, i.e. the last local register of the sequence S . As mentioned in the prior section, only the dead registers at the right end of local-registers can be overlapped. Translating into the superimposed graph notation, only the edges that are part of the path so far are counted. A heavy line does not appear between $r48$ and $r45$.

From the above example, we infer that we can calculate the gain of a register reassignment using a corresponding sequence on a superimposed graph. The superimposed graph, as shown in Figure 5c, is a multigraph that has multiple edges between two nodes. To simplify the graph, we construct a sequence graph $G(V, E)$, where $V = D_1 \cup D_2 \cup \dots \cup D_k$, and $E = \{(u, v) : u \in D_i \text{ and } v \in D_i, 1 \leq i \leq k\}$. The weight of a node u , $w(u)$, is the sum of the weights from all the complete graphs the node u belongs to, where weights of complete graphs are determined by the weights of their associated dead sets. An edge (u, v) has an annotation, denoted by $a(u, v)$, which is set of all dead sets where the edge belongs. Figure 5d shows an example of a sequence graph composed from the dead sets listed in Figure 5a. Recall that $w(D)$ denotes the weight of a dead set D .

On sequence graphs, we need to find a *node-sequence* that results in maximum gain. We formally define a node-sequence as follows.

Let a sequence of nodes $NS = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_q$, ($q \leq |V|$) be *node-sequence* from a sequence-graph $G(V, E)$. NS has the following properties.

- (1) $\forall 1 \leq i \leq q - 1$, edge $(v_i, v_{i+1}) \in E$ and,
- (2) The gain of NS is defined as

$$\text{gain of } NS = w(v_1) + \sum_{i=1}^{q-1} w(A_i \cap a(v_i, v_{i+1})) \quad (2)$$

$$\text{where } \begin{cases} A_1 = U & \text{(i.e. set of all given dead sets)} \\ A_{i+1} = A_i \cap a(v_i, v_{i+1}), & i = 1, \dots, q - 1 \\ w(A) = \sum_{D \in A} w(D), & \text{for any set } A \text{ of dead sets} \end{cases}$$

The first property is a trivial condition that ensures that the sequence forms a path on $G(V, E)$. The second property defines the gain of a node sequence. Each dead set can have a path composed of nodes in its complete graph. The definition implies that only the dead sets whose paths begin with the starting node of the node sequence contribute to the gain. In other words, the *gain* of an NS in a sequence graph is the sum of the weights of each path, which is part of NS in each complete graph.

Algorithm 1 `reassign_register()`**Input :** Dead sets, D_1, D_2, \dots, D_k , for all call sites in a procedure P .**Output :** A sequence of nodes.

```

1:  $G(V, E) \leftarrow$  sequence graph composed from  $D_1, D_2, \dots, D_k$ .
2:  $NS(SV, SE) \leftarrow \phi$  /* node sequence start with empty graph */
3:  $(u, v) \leftarrow$  edge with the biggest gain,  $(u, v) \in V, w(u) \geq w(v)$ 
4:  $SV = SV \cup \{u, v\}, SE = SE \cup \{(u, v)\}$  /* insert  $(u, v)$  to  $NS$  */
5:  $A \leftarrow a(u, v)$ 
6:  $t \leftarrow v$ 
7: while  $SV$  contains less than  $|V|$  nodes do
8:   Find  $(t, v)$  which has the biggest  $w(A \cap a(t, v))$ , where  $(s, t) \in SE$  and  $v \notin SV$ .
9:   if no such  $(t, v)$  exists then
10:    break
11:   end if
12:    $SE = SE \cup \{(t, v)\}, SV = SV \cup \{v\}$  /* append  $(t, v)$  to  $NS$  */
13:    $A \leftarrow A \cap a(t, v)$ 
14:    $t \leftarrow v$ 
15: end while
16: return reverse(NS) /* put the heaviest weight node at the end of the sequence */

```

To find an optimal node-sequence, we select a sequence of maximum gain among all such sequences. Since the overlapped dead registers are relevant to the gain, we can focus only on dead registers at each call site. Once we find such a node-sequence of dead registers on a sequence graph, we reverse it as we select the right-most dead register first for the node-sequence. We then arbitrarily build a sequence holding the rest of the local-registers. We finally concatenate the two sequences, putting the sequence of the dead registers in the back. The resulting sequence can now be used to guide the register reassignment. In Section 4, we will describe how to find a maximum gain node-sequence from sequence graphs using a greedy approach. Formal proof of the equivalency between finding an optimal register reassignment and finding a maximum gain node-sequence is also given in the Appendix.

4. ALGORITHM

We propose an algorithm to find the optimal register reassignments defined in Section 3. Our approach to solve this problem is to find the maximum gain nodesequences on the sequence graphs introduced in Section 3.2. An efficient heuristic is detailed in the following sections.

4.1 `reassign_register` and `multi_alloc_insert`

Our proposed algorithm, called `reassign_register`, is based on a greedy approach. The input to `reassign_register` is the dead sets (D_1, D_2, \dots, D_k) of a procedure. From D_1, D_2, \dots, D_k , we build a corresponding sequence graph G and then construct a node-sequence by appending one edge at a time. Our algorithm starts with a node-sequence NS that contains only one edge (u, v) whose gain in Eq. (2) is the greatest among all edges in G . At each iteration of the *while* body, we examine all the adjacent nodes to the end node of the NS so far and pick the node that increases the gain of NS most. This is described in line 8 of Algorithm 1. Note that we preserve Eq. (2) for computing the gain of a

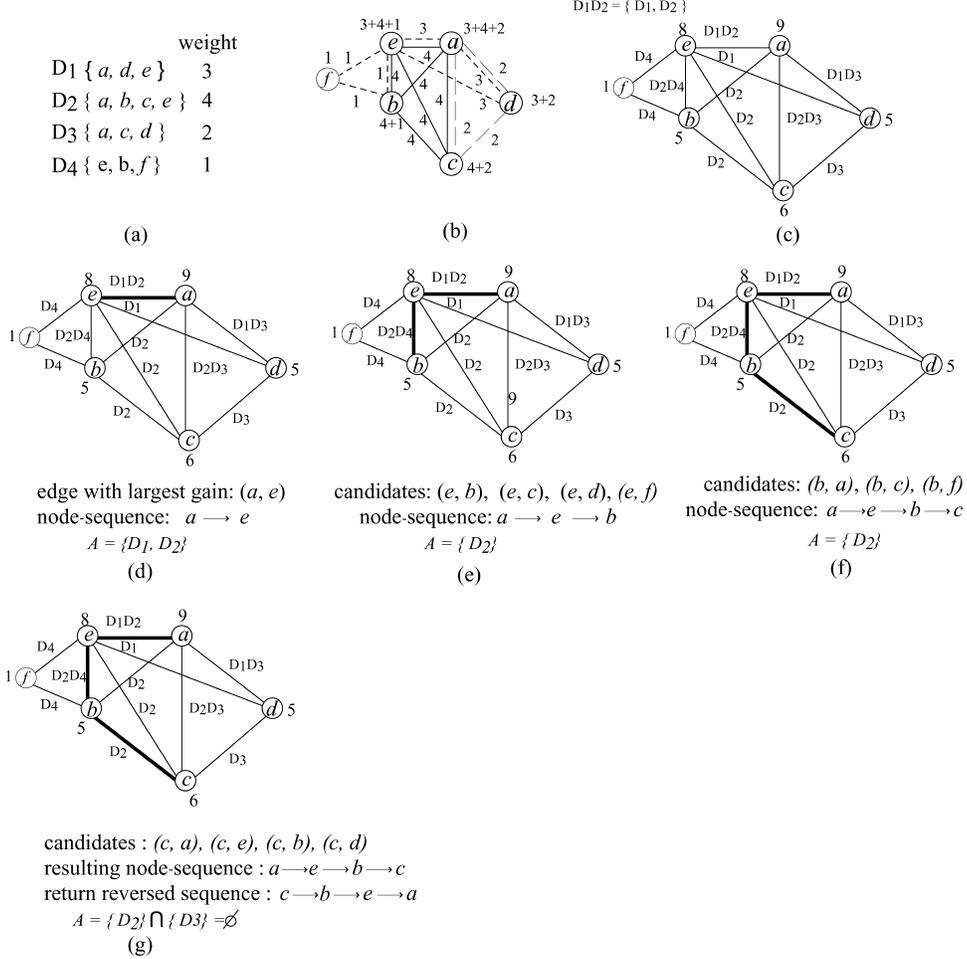


Fig. 6. An example illustrating each stage of edge addition to build a node-sequence.

node-sequence during iterations. Thus, the resulting gain of Algorithm 1 equals the gain of the original problem in Eq. (1). Algorithm 1 contains a sketch of `reassign_register`.

Figure 6 shows an example of `reassign_register` step-by-step. In Figure 6a, there are four distinct dead sets. The registers in each dead set are given simple names, such as a, b, \dots, f , for ease of description. The number after a dead set indicates the weight of the dead set. Figure 6b shows the superimposition of four complete graphs from D_1, D_2, D_3 , and D_4 , while Figure 6c shows the resulting sequence graph. Figure 6d through g illustrate each step to find the maximum gain node-sequence. At first, edge (a, e) is chosen since it has the biggest gain, $16 (= w(a) + w(U \cup a(a, e)) = 9 + (3 + 4))$. A is $a(a, e) = \{D_1, D_2\}$. The node-sequence is $a \rightarrow e$ since $w(a) \geq w(e)$. Then, we examine all nodes adjacent to node e , i.e., b, c, d , and f . For b , $(A = a(a, e)) \cap a(e, b) = \{D_1, D_2\} \cap \{D_2, D_4\} = \{D_2\}$, and $w(\{D_2\}) = 4$. Similarly, for c , $w(A \cap a(e, c)) = w(\{D_2\}) = 4$, and for

Algorithm 2 multi_alloc_insert()

input : Optimized code of a procedure P after register allocation
output : Register stack optimized code with multiple *alloc* instructions.

- 1: Let \mathcal{D} be the set of all dead sets in P , $\mathcal{D} = \phi$.
- 2: **for all** call site C in P **do**
- 3: Calculate dead set D for call site C .
- 4: **if** $D \in \mathcal{D}$ **then**
- 5: $w(D) = w(D) + w(C)$ /* $w(C)$ denotes the weight of call sites C . */
- 6: **else**
- 7: $w(D) = w(C)$
- 8: $\mathcal{D} = \mathcal{D} \cup \{D\}$
- 9: **end if**
- 10: **end for**
- 11: $\widehat{NS} = \text{reassign_register}(D)$ /* \widehat{NS} is a reversed node-sequence */
- 12: Let L be set of local registers of P .
- 13: $S \leftarrow$ arbitrarily concatenate all nodes in $L - V(\widehat{NS})$ before \widehat{NS} .
- 14: Reassign local registers used in P using S .
- 15: **for all** call site C in P **do**
- 16: Let D be the dead set of call site C .
- 17: **if** decide to resize based on \widehat{NS} , D **then**
- 18: Insert *alloc* and copy parameters. /* Resize register stack before and after
- 19: C using \widehat{NS} , D */
- 20: **end if**
- 21: **end for**

d , $w(A \cap a(e, d)) = w(\{D_1\}) = 3$, respectively. For f , $A \cap a(e, f) = \phi$. Among the two biggest gain nodes b and c , we arbitrarily pick b . Now the node sequence is $a \rightarrow e \rightarrow b$, and its gain is $16 + 4 = 20$. $A = A \cap a(e, b) = \{D_2\}$. In the next step, at b , among adjacent nodes a , c , e , and f , we pick c since $A \cap a(b, c) = \{D_2\}$ and a and e are already in the node sequence so far. The node sequence is now $a \rightarrow e \rightarrow b \rightarrow c$ (gain = $24 = 20 + w(\{D_2\})$) and $A = \{D_2\}$. Finally, at c our algorithm stops because a , e , b are already in the node-sequence and $A \cap a(c, d) = \phi$. The resulting node-sequence is $a \rightarrow e \rightarrow b \rightarrow c$, and its gain is 24.

Note that we can come up with a simpler heuristic than `reassign_register()` in which we only consider how many times a register appears in the dead sets (namely, node frequency). We sort nodes in the all dead sets in nonincreasing order of node frequencies, and then the result will produce a sequence of nodes. With this heuristic, the example in Figure 6 can result in $a \rightarrow e \rightarrow c \rightarrow d \rightarrow b \rightarrow f$. The gain of this sequence is 20 ($= 9 + 7 + 4$), which is less than 24, the gain we can obtain from `reassign_register()`. This simple heuristic has some advantages over `reassign_register()`. It does not build a sequence graph, but simply counts the frequency of each node. As a result, it has lower time complexity than that of `reassign_register()`, but generally produces less optimized register reassignments.

Algorithm 2 shows how we use the multi-alloc method. The optimized code of a procedure after register allocation is given as an input. First, we gather all the dead sets in a procedure, assigning a weight to each dead set. We can treat all call sites equally and give the same weight to all the corresponding dead

sets. Alternatively, we can use profiling information to weight each dead set in favor of its relative importance, like the frequency of calls of the associated call site. The weight of a dead set becomes the weights of the nodes in the dead set. With weighted dead sets, we obtain register reassignment for local-registers from `reassign_register()`. After we interchange local-register names by the register reassignment, we insert multiple `alloc` instructions wherever they are necessary. The `alloc` instructions inserted to resize the register stack frame also require the copying of the procedure parameters from the old out-registers to the new out-registers since we change the boundaries of local-registers and out-registers.

4.2 Algorithm Efficiency

We implement sequence graphs $(G(V, E))$ as adjacent lists. We represent annotations on edges as bit vectors where every dead set is encoded with one bit. Then, given dead sets, D_1, D_2, \dots, D_k , computing the gain of an intersection is able to run in time $O(k)$. Thus, for each iteration of `while` body of Algorithm 1, we are able to compute the node with the biggest gain in $O(k \cdot |V|)$, asymptotically. Since $O(|V|)$ times of iterations occur, the while loop takes $O(k \cdot |V|^2)$ time, in total. In constructing a sequence graph, to superimpose each dead set's complete graph, we first implement a sequence graph as an adjacent matrix and then convert it to the corresponding adjacent list. It takes $O(k \cdot |D_{\max}|^2 + |V|^2)$, where D_{\max} is the dead set of maximum cardinality among D_1, D_2, \dots, D_k . We adopt this approach to tackle both the building of a sequence graph and the finding of a node sequence efficiently. Overall, the time complexity of `reassign_register` is bound by $O(k \cdot |D_{\max}|^2 + k \cdot |V|^2)$.

5. EXPERIMENTAL RESULTS AND ANALYSIS

5.1 Experimental Environments

We perform our experiments on an Intel Itanium2 box with two 1.4GHz Itanium2 processors and a 1.5M L2 cache, running a 64bit version of RedHat Linux. We used the Open Research Compiler(ORC) with base optimization, which employs Itanium specific optimizations, such as if-conversion, data and control speculation, predicate analysis, and global instruction scheduling with resource management, etc., including, all classic optimizations [Intel Corporation and Chinese Academy of Sciences 2002]. We implemented our multi-alloc method that utilizes `reassign_register` described in Section 4 in ORC after its last register allocation. We also implemented multiple alloc optimization without considering the reassignment of local registers as in Settle et al. [2003], for the purpose of comparison. Their multi-alloc scheme also resizes the register stack frame before the call sites. Nonlive registers across calls are overlapped with the callee's register stack frame. Register liveness analysis is used to distinguish those registers that can be overlapped. However, only the nonlive registers at the end of the local-register region can be overlapped. Our technique also utilizes multiple alloc instructions to resize the register stack frame before the call sites, but our technique pushes more nonlive registers toward the end of

the local-register region by reassigning registers in order to force more dead registers to overlap with the callee's register stack frame. For convenience, we call our scheme RR-multi-alloc, which stands for *register reassigning multi-alloc*, while we call Settle et al.'s multi-alloc. We used the SPEC CINT2000 benchmarks to evaluate the effectiveness of our approach.

5.2 Implementation Details

To resize the register stack at a call site, two *alloc* instructions are required before and after a call site. In addition, procedure parameters from a caller to callee should be reassigned before the call site. Refer to Figure 4, which demonstrates the reduction of the register stack height. Output registers *r39* and *r40* holding parameters from function *f* to *g* in Figure 4a are reassigned to *r37* and *r38*, respectively, as in Figure 4b. Currently, we implement these reassignments by *mov* instructions. Some *nop* instructions are also added for bundling the *alloc* and *mov* instructions. Those *alloc* and *mov* instructions are pure overheads. To avoid the overheads' hiding the benefit from the reduction in the register stack height, we used thresholds. If the number of overlapped registers falls below a certain number or the number of reassigned output registers is above another certain number, we do not multi-alloc the call site. For our experiments, our threshold was 3 to 4 for the number of overlapped registers and 0 to 6 for the output procedure parameters. By through data-flow analysis of the live ranges of registers, the number of the *allocs* augmented in a procedure (i.e., a caller) can be minimized. Also, the *mov* for reassigning procedure parameters and *nop* for bundling instructions can be minimized if our register reassignment more fully cooperates with the instruction selection and the register allocation.

Note that our optimization can be profile-guided by giving the more weight to the more significant call sites when we build a sequence graph. We can take a couple of approaches in deciding the relative importance of each call site. In the simplest way, the frequency of the calls of a call site can be used as the weight. Alternatively, if it is possible to pinpoint where the overflow occurs, we can utilize the chain of the calls that causes the overflow for the purpose of differentiating call sites in a procedure. At this time, our experiments are not based on profiling information, and all call sites in a procedure have the same weight.

5.3 Characteristics of Benchmarks

5.3.1 Call Sites Characteristics. Table I lists the static and dynamic characteristics of the call sites of SPEC CINT2000 programs. The second and third columns show static numbers of the total call sites and average numbers of call sites per procedure, respectively. The fourth column shows the average percentages of call sites per procedure where RR-multi-alloc is applied. We found SPEC CINT2000 benchmarks have about nine static call sites in a procedure on average. When our RR-multi-alloc is used, about 40% of those call sites resize default register stack frames. One extreme case is *crafty*, which has about 30 call sites in a procedure, and which is about three times more than average.

Table I. Characteristics of SPEC CINT2000: Numbers of Procedure Call Sites in Program and Per Procedure

Benchmarks	(Static) in program	(Static) per procedure	(Static) alloc-site percentage	(Dynamic) alloc-site percentage	(Dynamic) maximum call depth
gzip	254	4.98	40%	0.2%	12
vpr	1514	8.96	47%	1.7%	11
gcc	17822	10.80	40%	41.8%	35
mcf	54	6.75	36%	0.2%	34
crafty	405	29.42	55%	83.0%	33
parser	1214	5.62	30%	24.4%	73
perlbnk	5775	6.72	40%	10.2%	54
gap	3576	6.62	47%	10.5%	362
vortex	8097	9.32	37%	28.4%	28
bzip2	254	5.52	39%	0.0%	8
twolf	611	8.04	45%	21.8%	9
Avg.	3598	9.34	41%	20.0%	60

Our multi-alloc scheme with register reassignment resizes the register stack frame for more than one half of those call sites.

The fifth column shows the dynamic percentage of calls where RR-multi-alloc is applied. We traced the call sites of the entire execution in support of GNU compiler toolchain’s function instrumentation. Then, we counted the call sites matched by ORC’s multi-allocated call sites. Unlike the static percentages between 30 to 55% over all benchmarks, the dynamic percentages of alloc-site show wide variations among benchmarks. Benchmarks with high dynamic percentage, such as *gcc*, *crafty*, *twolf*, and *vortex*, show some improvement in performance (see later, Table V and Figure 9). Although *vpr* and *mcf* showed up to 4 and 1.5% improvement, their dynamic percentages of alloc-site are low. These low percentages are partly from the loss of information. Gcc generates instrumentation calls just after a function entry and just before the function exit in the callee’s body, while *alloc* instructions are added before and after a call site in the caller’s body. For *vpr* and *mcf*, many of the allocated call sites are calls to library functions, which are not captured by Gcc’s instrumentation. Finally, the sixth column shows the maximum call depth of each benchmark.

5.3.2 Stacked Registers Characteristics. Figure 7 shows the characteristics of the stacked registers of SPEC CINT2000 programs. The first bar represents the average numbers of stacked registers allocated in a procedure (*in/local/out-registers*). The second bar shows the average numbers of dead stacked registers (*in/local/out-registers*) that are unused before at least one call site. The third bar shows the average numbers of dead *local-registers* among those total dead stacked registers (*in/local/out-registers*) shown in the second bar. These dead *local-registers* are the potential candidates for overlapping to resize stack frames. All the results are obtained with the ORC using base optimization. The total number of stacked registers is about 12, which is far less than the allowed number of registers, 96. Among the stacked registers, we found about six registers that are dead at the call sites. Although one-half of the stacked registers are dead at the call sites, four or five registers out of six dead registers are

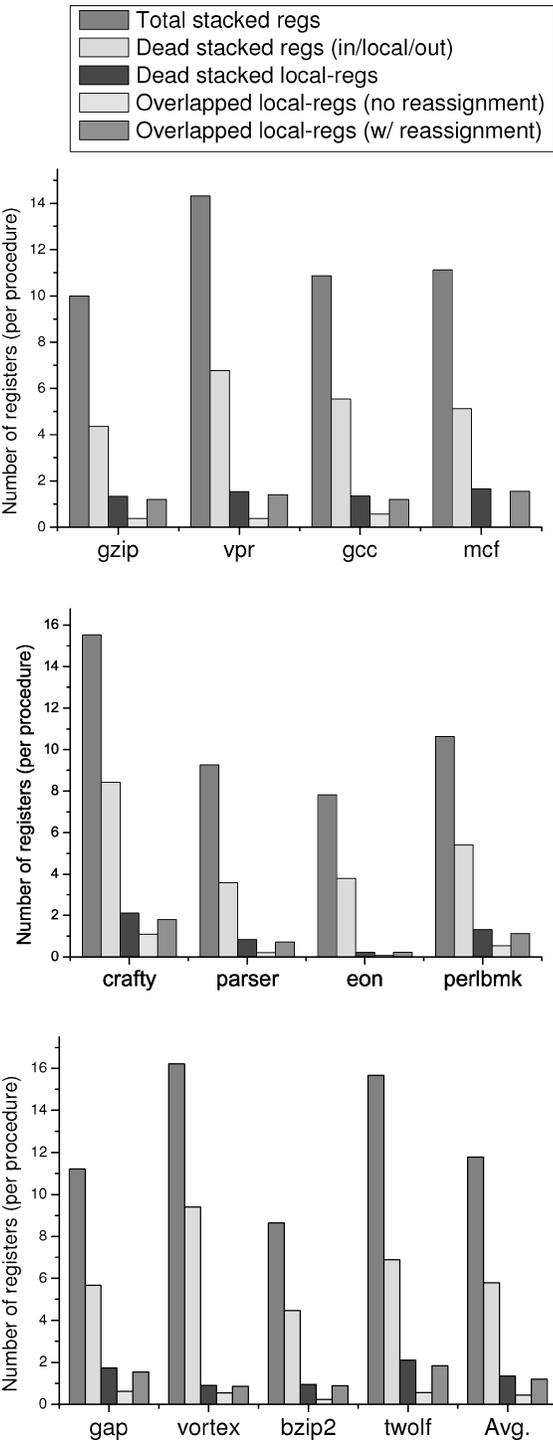


Fig. 7. Static numbers of registers per procedure for SPEC CINT2000.

Table II. Run-Time Performance: RSE Cycles

Benchmarks	Base (% of RSE cycles in total cpu cycles)	Multi-alloc	RR-multi-alloc
gzip	8.918×10^6 (0.003%)	8.723×10^6	8.176×10^6
vpr	2.672×10^7 (0.008%)	2.647×10^7	2.234×10^7
gcc	6.629×10^9 (2.885%)	5.602×10^9	4.197×10^9
mcf	4.579×10^7 (0.003%)	4.576×10^7	4.575×10^7
crafty	8.587×10^9 (4.841%)	8.577×10^9	7.692×10^9
parser	2.052×10^9 (0.400%)	2.509×10^9	1.996×10^9
perlbmk	4.883×10^9 (1.295%)	5.805×10^9	5.760×10^9
gap	3.537×10^8 (0.130%)	3.295×10^8	3.466×10^8
vortex	3.027×10^9 (0.854%)	2.767×10^9	2.668×10^9
bzip2	1.898×10^8 (0.058%)	1.899×10^8	1.897×10^8
twolf	5.929×10^9 (0.972%)	5.900×10^9	5.927×10^9

out-registers, which already overlap with the callee’s register stack frame. Only one or two registers are dead among the *local-registers*. The register allocator in ORC is an integration of the approaches of Chow [Chow 1988; Chow and Hennessy 1984] and Chaitin-Briggs [Chaitin 1982; Briggs et al. 1994]. Chow’s global allocator utilizes cost and saving estimates for assigning a variable into a register within a local code block in deciding on a node to color. Caller-saved and callee-saved registers each have their own respective merits under different circumstances. Chow’s register allocator computes different priorities with respect to the register classes and assigns each program variable to the best register class [Chow 1988]. The register allocator in ORC classifies stacked registers into stacked callee-save and stacked caller-save. The former is used for live registers across call sites and is allocated in *in/local-register* regions. Meanwhile, the latter is mainly for a scratch pad, not live across call sites and allocated in *out-register* region. Hence, the current register allocator in ORC already, to some degree, overlap not-live (dead) registers with the next register stack frame by assigning them in the *out-register* region. On top of the ORC register allocation, our technique overlaps more dead registers in the local-registers region with the callee’s register stack frame by reassigning the local-registers.

The fourth and fifth bars show the average numbers of local-registers which overlap with the callee’s register stack frame using multi-alloc and RR-multi-alloc, respectively. The multi-alloc scheme overlaps only a one-third of the dead local-registers. Our sophisticated RR-multi-alloc scheme, which exploits register reassignment considering multiple call sites, can overlap 90% of dead local-registers.

5.4 Runtime Performance

5.4.1 Reduction in RSE Overhead.

Using the performance monitoring unit in the Itanium architecture, we measured the cpu cycles spent due to RSE. Table II compares RSE cycles using the ORC with base optimization, multi-alloc, and our RR-multi-alloc. The second column presents RSE cycles when we use the base ORC. The ratio of those RSE cycles to the total cpu cycles is shown in parentheses. The corresponding RSE cycles by multi-alloc and RR-multi-alloc

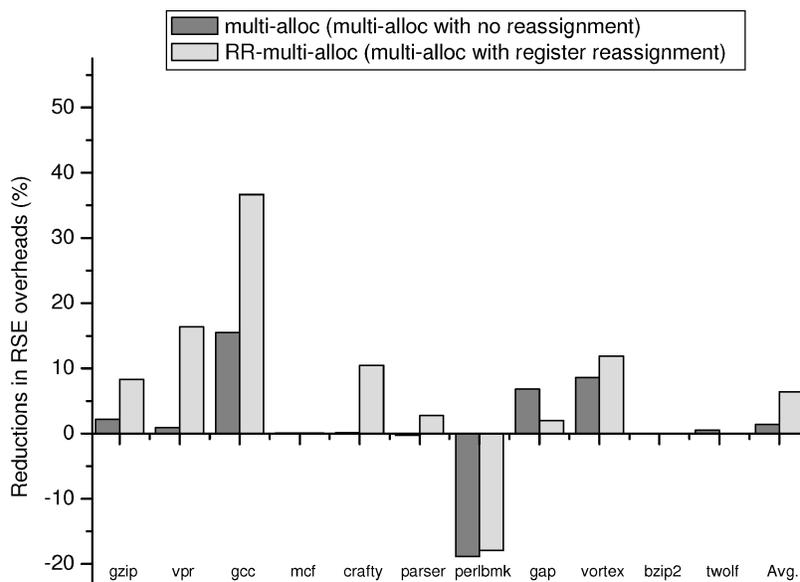


Fig. 8. The percentage of reduction in RSE cycles when multi-alloc and RR-multi-alloc are used, respectively.

are shown in the third and fourth columns, respectively. Figure 8 illustrates the reductions of the RSE cycles in percentages compared with the RSE cycles from the base ORC.

Our RR-multi-alloc results in sizable reduction in RSE cycles for every benchmark program, besides *perlbmk*. Furthermore, RR-multi-alloc achieves a three times larger reduction than multi-alloc on average. Particularly, for *vpr* and *crafty*, RR-multi-alloc reduces RSE cycles by 16 and 10%, respectively, while multi-alloc reduces RSE cycles only by 0.9 and 0.1%. For *gcc*, though multi-alloc already results in 15% reductions, RR-multi-alloc boosts the reductions up to 37%.

For *perlbmk*, both RR-multi-alloc and multi-alloc increase the RSE overhead. For *gap* and *twolf*, RR-multi-alloc reduces the RSE overhead less than multi-alloc. Generating more *alloc* instructions does not always lead to reductions in register overflow/underflow since the reduction of register stack heights in one part of a call sequence can unfortunately cause the increase of register overflow/underflow in other parts of the sequence. In addition, the RSE hardware can exploit the explicit register stack frame information to eagerly spill and fill registers from the register stack to memory at the best opportunity independent of the calling and called procedures [Intel Corporation 2002]. We think that such implicit RSE behavior and the unfortunate situation described above may cause the result for *perlbmk*, *gap*, and *twolf*.

For all benchmark programs, RR-multi-alloc and multi-alloc reduce RSE cycles on average by 6.4 and 1.4% points, respectively.

5.4.2 Increase in the Number of Total Instructions. We also measured the number of retired instructions using the performance monitoring unit of the

Table III. Numbers of Retired IA64 Instructions (Including NOP)

Benchmarks	base	multi-alloc	RR-multi-alloc	increase over ORC (%)	
				multi-alloc	RR-multi-alloc
gzip	5.959×10^{11}	5.959×10^{11}	5.959×10^{11}	0.001%	0.001%
vpr	3.429×10^{11}	3.431×10^{11}	3.433×10^{11}	0.059%	0.102%
gcc	3.031×10^{11}	3.043×10^{11}	3.075×10^{11}	0.417%	1.448%
mcf	1.364×10^{11}	1.364×10^{11}	1.364×10^{11}	0.000%	0.000%
crafty	3.059×10^{11}	3.059×10^{11}	3.066×10^{11}	0.000%	0.227%
parser	6.844×10^{11}	6.869×10^{11}	6.880×10^{11}	0.369%	0.517%
perlbmk	7.986×10^{11}	7.986×10^{11}	7.987×10^{11}	0.000%	0.018%
gap	4.397×10^{11}	4.399×10^{11}	4.404×10^{11}	0.053%	0.155%
vortex	5.514×10^{11}	5.527×10^{11}	5.517×10^{11}	0.252%	0.069%
bzip2	5.333×10^{11}	5.333×10^{11}	5.333×10^{11}	0.000%	0.000%
twolf	7.467×10^{11}	7.467×10^{11}	7.467×10^{11}	0.001%	0.003%
Avg.				0.105%	0.231%

Table IV. Numbers of Retired NOP Instructions

Benchmarks	base	multi-alloc	RR-multi-alloc	increase over ORC (%)	
				multi-alloc	RR-multi-alloc
gzip	1.349×10^{11}	1.349×10^{11}	1.349×10^{11}	0.003%	0.003%
vpr	1.072×10^{11}	1.073×10^{11}	1.074×10^{11}	0.104%	0.201%
gcc	9.335×10^{10}	9.421×10^{10}	9.624×10^{10}	0.927%	3.100%
mcf	4.571×10^{10}	4.571×10^{10}	4.571×10^{10}	0.000%	0.000%
crafty	7.019×10^{10}	7.019×10^{10}	7.073×10^{10}	0.000%	0.768%
parser	2.222×10^{11}	2.223×10^{11}	2.224×10^{11}	0.786%	1.096%
perlbmk	1.945×10^{11}	1.945×10^{11}	1.946×10^{11}	-0.003%	0.061%
gap	1.340×10^{11}	1.342×10^{11}	1.345×10^{11}	0.131%	0.370%
vortex	1.430×10^{11}	1.438×10^{11}	1.423×10^{11}	0.571%	-0.421%
bzip2	1.421×10^{11}	1.421×10^{11}	1.421×10^{11}	0.000%	0.000%
twolf	3.218×10^{11}	3.218×10^{11}	3.218×10^{11}	0.002%	0.004%
Avg.				0.229%	0.471%

Itanium architecture. Table III presents the numbers of retired instructions, including NOP, for base ORC, multi-alloc and RR-multi-alloc. Table IV shows the numbers of retired NOP instructions. In order to resize the register stack frame for one call site, two *alloc* instructions are needed before and after the call site. Besides *alloc* instructions, a few *mov* instructions are needed before each *allocated* call site for the purpose of copying procedure parameters to the new end of the resized register stack frame from the original end of the default one. Some *nop* instructions are also added for bundling those *alloc* and *mov* instructions. Thus, both RR-multi-alloc and multi-alloc usually execute more instructions than base ORC. Comparing RR-multi-alloc and multi-alloc, RR-multi-alloc usually executes more instructions than multi-alloc since it inserts more *allocs* to resize the register stack frame than multi-alloc (recall Figure 4 in which some call sites are *allocated* after a proper register reassignment that cannot be *allocated* originally). Note that the number of added *mov* and/or *nop* instructions for a call site is determined solely by the number of procedure parameters and the bundling capability of the scheduler. Reassigning registers (RR-multi-alloc) does not incur more *mov* and/or *nop* instructions for the same call site than multi-alloc.

Table V. Run-Time Performance: Total CPU Cycles

Benchmarks	base	multi-alloc	RR-multi-alloc
gzip	2.699×10^{11}	2.693×10^{11}	2.693×10^{11}
vpr	3.521×10^{11}	3.380×10^{11}	3.377×10^{11}
gcc	2.298×10^{11}	2.253×10^{11}	2.240×10^{11}
mcf	1.315×10^{12}	1.301×10^{12}	1.298×10^{12}
crafty	1.774×10^{11}	1.776×10^{11}	1.736×10^{11}
parser	5.068×10^{11}	5.099×10^{11}	5.079×10^{11}
perlbmk	3.770×10^{11}	3.808×10^{11}	3.774×10^{11}
gap	2.719×10^{11}	2.684×10^{11}	2.678×10^{11}
vortex	3.534×10^{11}	3.507×10^{11}	3.474×10^{11}
bzip2	3.274×10^{11}	3.274×10^{11}	3.234×10^{11}
twolf	6.101×10^{11}	5.889×10^{11}	5.834×10^{11}

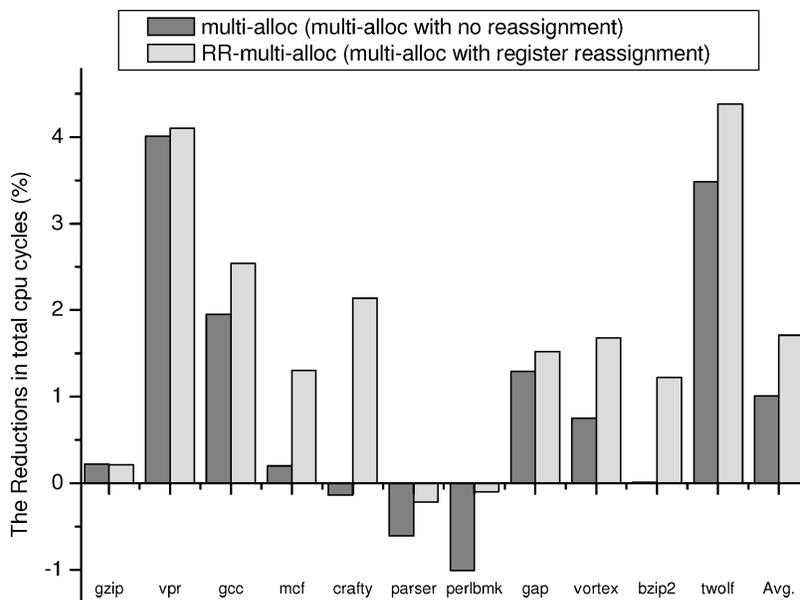


Fig. 9. The percentage of reduction in the total cpu cycles when multi-alloc and RR-multi-alloc are used, respectively.

5.4.3 Reduction in the Total cpu Cycles. We also measured the total execution cycles using the performance monitoring unit of Itanium architecture. Table V shows the performance of base ORC, multi-alloc, and RR-multi-alloc in terms of total cpu cycles. Figure 9 illustrates the percentage of reductions in the total cpu cycles of multi-alloc and RR-multi-alloc over base ORC, respectively. Our RR-multi-alloc improves the performance of every benchmark program, except for *parser* and *perlbmk*. For most programs, the performance gains are more than the percentage of the RSE cycles from the base ORC, which are shown in Table II. According to the cycle breakdowns, both multi-alloc and RR-multi-alloc reduce not only RSE cycles, but also other portions of the cycle breakdown, such as data access latency and back-end stalls because of front-end stalls. We think the sharing of limited memory bandwidth between

the register stack overflow/underflow and data access causes the reduction in other portions of cycle breakdown. We improve performance by combining the reduction in RSE cycles and the reductions in other portions of the cycle breakdown.

Vpr and *twolf* show great performance improvements for both RR-multi-alloc and multi-alloc, although their percentages of RSE cycles are quite low. These improvements are largely due to the reductions in front-end stalls. For *crafty*, which has the highest RSE percentage among all programs at 4.8%, RR-multi-alloc achieves a performance gain of 2.14%. Reductions in RSE cycles, data-access stalls, and back-end stalls because of front-end stalls jointly contribute to the reduction in the total cpu cycles. On the other hand, multi-alloc slightly increases total cpu cycles by 0.14% due to the increase in number of retired instructions and an increase in other portions of the cycle breakdown. For *crafty*, register reassignment greatly affects reduction in register overflow/underflow. For *gcc*, as a whole, a 2.54% reduction in the total cpu cycles is achieved with our RR-multi-alloc. Referring to Tables II and V, around 40% of that total reduction is from the reduction in RSE cycles. RR-multi-alloc also outperforms multi-alloc, which reduces the total cpu cycles by 1.95%. For *parser*, an increase in the numbers of instructions for instrumenting call sites together with a low percentage of RSE cycles in the total cpu cycles causes overall performance degradation (refer to Tables III and IV for increases in instructions). The increase in cpu cycles for *perlbmk* is due to an increase in RSE cycles and the number of retired instructions. Overall, RR-multi-alloc reduces the total cpu cycles 1.71%, on average, while multi-alloc reduces the total cpu cycles 1.01%, on average.

6. RELATED WORKS

Douillet et al. [2002] first proposed a multi-alloc method to minimize the overhead associated with RSE spill/fill. They analyzed the stack register usage per basic block and captured the multiple control flows within a procedure that use different numbers of stacked registers. By inserting *alloc* into multiple places in a procedure and resizing the register stack frame differently for different paths, they could reduce the stacked register usage along some paths. Their scheme, however, did not yield satisfactory results.

Settle et al. [2003] proposed another multi-alloc scheme that resizes the register stack frame before call sites in order to reduce the total height of the stack frames over all procedures in the call stack. The key idea is nonlive registers across calls overlap with the callee's register stack frame. Register liveness analysis is used to distinguish registers that can be overlapped. Although, Settle et al. provided a great deal of useful evidence that an overlapping register stack frame would lead to RSE overhead reduction, their scheme only takes advantage of the nonlive registers at the end of the local-register region.

Hoflehner et al. [2004] discuss how the register stack optimization technique attributes to the performance of on-line transaction processing (OLTP). They implemented the multi-alloc method of Settle et al. [2003] in the Intel C/C++

Itanium compiler. They report that the RSE optimization reduced RSE spills/fills by about 4% and increased throughput by 1.15%.

Weldon et al. [2002] examined various RSE optimizations techniques with dynamic register usage and dead register evaluation information because of imbalanced paths of applications' execution. They simulated heap-based RSE implementation, unlike the original stack-based RSE in the Itanium architecture, in order to fully take advantage of that profile information, which assumed nonoverlapping frames between the callers and callees.

Yang et al. [2003] proposed a method that decides upon the appropriate quota of stacked registers for each procedure. Their approach is not based on the multi-alloc method, but has the same goal to reduce the overhead of the RSE spill/fill. Yang et al. analyzed a call graph and used a cost model in order to find the trade-off between explicit register spills and the usage of stacked registers managed by the RSE. They observed that reducing the stacked register usage in some procedures could reduce the total memory access time of spilling registers. Using their quota allocation method, *perlbmk*, which originally spent 23% of the total execution time in RSE, spent around 0% of the cycles in the RSE. They also reported a 13% performance improvement for *perlbmk*.

Our method presented in this paper follows the multi-alloc method used in the work of Settle et al. However, our work is different in that we have devised a register reassignment scheme to maximize the overlap with the callee's register stack frame. In particular, our scheme considers multiple call sites within a procedure for the overall minimization of the register stack height. Moreover, we defined the problem of finding optimal register reassignment and formulated it as a path-finding problem in a graph called a *sequence graph*. We also proposed an efficient heuristic algorithm to solve the problem.

7. CONCLUSION

Hardware managed register stacks or register windows are good design choices, considering that we have enough space to place a large number of physical registers on processor chips, but have only a limited number of virtual registers due to the encoding space on the instruction set architectures. For better support for a hardware managed register stack, we apply compiler-assisted register reassignment. Given that multiple *alloc* instructions are supported in a procedure, we can also freely resize the register stack frame multiple times within a procedure. Utilizing both register-reassignment and resizing the register stack frame, we achieve the maximal overlapping of the dead local registers between procedure call boundaries. Since our register-reassignment approach is applied after the register-allocation phase, we introduce minimal changes on existing compiler structures, widening the opportunities for any compilers to adopt our scheme for register stack optimizations.

According to our experiment with SPEC CINT2000, our approach reduces RSE cycles by 6.4% and the total cpu cycles by 1.7%, on average. For *crafty* and *gcc*, which have high RSE overheads of around 4.8 and 2.9%, respectively, when base ORC is used, our approach achieves considerable RSE cycle reductions by 10.4 and by 36.7%, along with total cycle reduction by 2.1 and 2.5%, respectively.

APPENDIX

We claim that the gain of a node-sequence on a sequence graph is the same as the gain of the register reassignment. Let D_1, D_2, \dots, D_k denote dead sets of a procedure and L be the set of all stack registers in local region. Then an instance of *overflow-minimizing register reassignment problem* is denoted as $(D_1, D_2, \dots, D_k, L, RA)$, where RA is a bijective register reassignment mapping from L to L . On the other hand, $(D_1, D_2, \dots, D_k, L, NS)$ denotes an instance of *maximum gain node-sequence finding problem in the sequence graph of D_1, D_2, \dots, D_k* .

Given an RA of $(D_1, D_2, \dots, D_k, L, RA)$, there is a sequence $S = s_1 \rightarrow \dots \rightarrow s_l$, ($l = |L|$) of all stacked registers in L by the definition of the register reassignment. Furthermore, for each D_i ($1 \leq i \leq k$) there is a suffix of S , denoted by $\text{suf}S_i = s_m \rightarrow \dots \rightarrow s_l$, where $s_m, \dots, s_l \in D_i$ and $s_{m-1} \notin D_i$ if s_{m-1} exists in S . Then, from Eq. (1), $\text{gain}(S, D_i) = \text{gain}(\text{suf}S_i, D_i)$, ($1 \leq i \leq k$). Let $\text{suf}S^{\max}$ be the longest among all $\text{suf}S_i$, $1 \leq i \leq k$. The implied NS of RA is the reversal of $\text{suf}S^{\max}$, denoted as $\overline{\text{suf}S^{\max}}$.

THEOREM A.1. *Given $(D_1, D_2, \dots, D_k, L, RA)$ and $(D_1, D_2, \dots, D_k, L, NS)$, if NS is implied from RA , the gain \mathcal{G} of RA in Equation (1) is the same as the gain of NS in Equation (2).*

PROOF. Suppose $\text{suf}S_i$, ($1 \leq i \leq k$) is $s_m \rightarrow s_{m+1} \rightarrow \dots \rightarrow s_{l-1} \rightarrow s_l$, ($1 \leq m \leq l$). By Equation (1), for $1 \leq i \leq k$,

$$\text{gain}(S, D_i) = \text{gain}(\text{suf}S_i, D_i) = w(D_i) \cdot (l - m + 1)$$

where $w(D_i)$ denotes the weight of a dead set D_i .

The reversion of $\text{suf}S_i, \overline{\text{suf}S_i}$, can be represented as a path in the complete graph of D_i . For every node and edge of the complete graph of D_i , its weight is $w(D_i)$. Thus,

$$\begin{aligned} & \text{(The sum of the weights of path } \overline{\text{suf}S_i} \text{ in the complete graph of } D_i) \\ &= w_i(s_l) + w_i(s_l, s_{l-1}) + \dots + w_i(s_{m+1}, s_m) \\ &= w(D_i) + w(D_i) + \dots + w(D_i) \\ &= w(D_i) \cdot (l - m + 1) \end{aligned} ,$$

where $w_i(v)$ denotes the weight of node v in the complete graph of D_i .

Thus, for all $1 \leq i \leq k$,

$$\text{gain}(S, D_i) = \text{(The sum of weights of path } \overline{\text{suf}S_i} \text{ in the complete graph of } D_i)$$

The sequence graph of D_1, D_2, \dots, D_k is composed from the superimposition of all complete graph of D_1, D_2, \dots, D_k . From Eq. (2) and the above result,

$$\begin{aligned} & \text{gain of } \overline{\text{suf}S^{\max}} \\ &= \sum_{i=1}^k \text{(The sum of weights of path } \overline{\text{suf}S_i} \text{ in the complete graph of } D_i) \\ &= \sum_{i=1}^k \text{gain}(S, D_i) \quad \square \end{aligned}$$

Conversely, the *implied RA of NS* in $(D_1, D_2, \dots, D_k, L, NS)$ is defined as follows: concatenate all registers in $L - V(NS)$ after NS in arbitrary order, where $V(NS)$ is the set of all nodes in NS ; let S be the reversal of the resulting sequence; implied *RA of NS* is determined from S by the definition of the register reassignment.

THEOREM A.2. *Given $(D_1, D_2, \dots, D_k, L, RA)$ and $(D_1, D_2, \dots, D_k, L, NS)$, if RA is implied from NS , the gain of NS is the same as the gain \mathcal{G} of RA .*

The proof of Theorem A.2 is similar to that of Theorem A.1 and is omitted. Theorems A.1 and A.2 indicate that we can solve the problem of overflow-minimizing register reassignment by translating it into the problem of finding a maximum gain node-sequence on a sequence graph.

ACKNOWLEDGMENTS

This research was supported by the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment). (IITA- 2005-C1090-0502-0031)

REFERENCES

- BRIGGS, P., COOPER, K., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems* 16, 3, 428–425.
- CHAITIN, G. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*. 201–207.
- CHOW, F. 1988. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*. 85–94.
- CHOW, F. AND HENNESSY, J. 1984. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*. 222–232.
- DOUILLET, A., AMARAL, J., AND GAO, G. 2002. Fine-grained stacked register allocation for the itanium architecture. In *15th Workshop on Languages and Compilers for Parallel Computing*.
- HOFLEHNER, G., KIRKEGAARD, K., SKINNER, R., LAVERY, D., AND LEE, Y. 2004. Compiler optimizations for transaction processing workloads on itanium linux systems. In *Proceedings of the 37th International Symposium on Microarchitecture*. 294–303.
- Intel Corporation 2002. *Intel Itanium Architecture Software Developer's Manual*. Intel Corporation, Santa Clara, CA.
- Intel Corporation and Chinese Academy of Sciences 2002. *Open Research Compiler for Itanium Processor Family (ORC version 2.1)*. Intel Corporation and Chinese Academy of Sciences, <http://ipf-orc.sourceforge.net>.
- KURLANDER, S. AND FISHER, C. 1996. Minimum cost interprocedural register allocation. In *Proceedings of the 23rd SIGPLAN-SIGACT Symposium on Principles of Programming Language*. 230–241.
- LUEH, G. AND GROSS, T. 1997. Call-cost directed register allocation. In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*. 296–307.
- SETTLE, A., CONNORS, D., HOFLEHNER, G., AND LAVERY, D. 2003. Optimization for the intel itanium architecture register stack. In *Proceedings of the International Symposium on Code Generation and Optimization*. 115–124.
- STEENKISTE, P. AND HENNESSY, J. 1989. A simple interprocedural register allocation algorithm and its effectiveness for list. *ACM Transactions on Programming Languages and Systems* 11, 1, 1–30.
- WALL, D. 1986. Global register allocation at link time. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*. 264–275.

- WEAVER, D. AND GERMOND, T. 1994. *The SPARC Architecture Manual*. SPARC International Inc., Menlo Park, CA.
- WELDON, R., CHANG, S., WANG, H., HOFLEHNER, G., WANG, P., LAVERY, D., AND SHEN, J. 2002. Quantitative evaluation of the register stack engine and optimization for future itanium processors. In *Proceedings of the Sixth Workshop on Interaction between Compilers and Computer Architectures*. Boston.
- YANG, L., CHAN, S., GAO, G., JU, R., LUEH, G., AND ZHANG, Z. 2003. Inter-procedural stacked register allocation for itanium like architecture. In *Proceedings of the 17th International Conference on Supercomputing*. 215–225.

Received November 2005; revised January 2006; accepted February 2006