Register Renaming Algorithm
for Register Stack Optimization

Yoonseo Choi and Hwansoo Han

CS/TR-2005-217
February 17, 2005

KAIST
Department of Computer Science
Abstract

Architectures with register stack can implement efficient calling conventions. By applying register renaming, register stack partially overlaps caller’s registers with callee’s registers. Using those overlapped registers, callers are able to pass parameters to callees without memory stack. Moreover, callee-save registers are simply renamed without saving in memory stack, which is a key performance boost for register stack. Most recent instance of register stack can be found in Intel Itanium architecture. A hardware called register stack engine (RSE) provides an illusion of infinite-length register stack using memory-backed handling of overflow and underflow for a physically limited number of registers. Despite such hardware support, some applications suffer from the overhead to handle register stack overflow and underflow. Previous research reported up to 24% of execution cycles are spent for the overhead [15]. In this paper we propose a compiler supported register renaming that interprocedurally reduces register stack height and minimizes the overhead involved in register stack overflow and underflow as a result. Live analysis traditionally used in register allocation is adopted to find the set of registers which are unnecessary to keep their values between procedure boundaries. Since a procedure contains many call sites, unnecessary registers to keep at each call site may all be different. By profiling register stack overflow and underflow, and giving appropriate weight for each call site, we find optimal register renaming that maximally reduces register stack height over all program execution. Using our technique we reduce up to 27% of register stack overhead for SPEC CINT2000 benchmarks.

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 to reduce the number of register spill/fill, which are intrinsically load/store operations. Typical register allocators focus on reduction of register spill/fill within a procedure boundary. Another important issue in register allocation, however, exists on 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 memory access penalty associated with procedure calls [12, 4, 11, 8, 9].
Microarchitectural solutions to this problem can be found in many commercial microprocessors. Providing large physical register file and maintaining them in register windows [13] or register stack [6] can mitigate the overhead of register save/restore around procedure boundaries. The SPARC architecture allocates a register window per procedure which consists of 24 registers [13]. First 8 registers are used for procedure arguments (in-registers). The next 8 registers are used for local registers for the current procedure (local-registers). The last 8 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 similar mechanism called register stack, but a special instruction can control the sizes of in/local/out-registers for each procedure [6]. Since the Itanium architecture controls register stack using instructions, they can allow much flexible usage of register stack. A special instruction (alloc) are used for this purpose, enabling dynamic resizing of register stack frame within a procedure.

Both register stack and register window offer a fast argument passing mechanism among procedures. More importantly, they can reduce the overhead of register save/restore on procedure calls that typically require memory accesses otherwise. Hardware-managed register stack and register windows are key components which provide the illusion of infinite length of registers by automatically backing up and reloading the contents of physical registers from the memory without explicit intervention of software. One of current trend in microprocessors is to equip with large physical registers, even though the number of visible registers from applications are fixed. Both register stack and register window can benefit from large physical register files, less frequently backing-up and reloading to and from the memory. We found that carefully allocating registers into proper regions among in/local/out-registers and dynamically resizing register stack frame can further avoid significant amount of register spill/fill overhead involved in register stack overflow and underflow. In this work, we propose a compiler supported register renaming technique to interprocedurally reduce over all register stack height by utilizing the software resizing feature of the register stack in Itanium-like architectures. Before further explanation of our technique, we give a short background of register stack and an overview of prior works. Along with explanation of register renaming as register stack height reduction, we present our profile-guided register renaming scheme and its algorithm. Finally, we present experimental results and conclude this paper.
2 Background and Related Work

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 rest 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 to use within the procedure. The actual number of physical registers for register stack are usually larger than the number of virtual stacked registers. Register stack consists of those physical registers and register stack engine (RSE). The stacked registers claimed by a procedure are mapped on to physical registers and we call them register stack frame for that procedure.

Each register stack frame is grouped into three parts: (1) in-registers holding incoming parameters from callers, (2) local-registers containing the registers current procedure uses during its lifetime, and (3) out-registers containing 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 register stack frame is determined at the beginning of procedure by invoking an special instruction, alloc.

\[
\text{alloc } \langle \text{targ\_reg} \rangle = \text{ar.pfs, in, local, out, rot}
\]

where \( \text{ar.pfs} \) is previous function state register to restore the previous state upon return. The total number of registers in the register stack frame for a procedure is \#\text{in-registers} + \#\text{local-registers} + \#\text{out-registers} \leq 96, with conditions of \#\text{in-register} \leq 8 and \#\text{out-registers} \leq 8.
To provide uniform accesses to its stacked register frame, the starting register of each register stack frame is renamed to r32 and the following registers are also renamed in order. The mapping between virtual registers and physical registers are illustrated in Figure 1, where procedure $f()$ calls $g()$. The parameters are passed from $f()$ to $g()$ through overlapping of out-registers of $f()$ and in-registers of $g()$. Register stack engine (RSE) manages in a circular manner the mapping of register stack frame to physical registers [6]. When a procedure is called, the current register stack frame is set to out-registers of caller and is resized by issuing alloc instruction. If multiple register stack frames use up total physical registers, RSE overflows some of the stacked registers to backing store in memory. When returning from a procedure, previous register stack frame needs to be restored. If there were any overflowed registers, RSE underflows the requested registers back to physical registers from the backing store. Note that alloc instruction is only responsible for sizing of current stack frame. Setting the start of current stack frame is done by RSE when call instruction is invoked. Thus, the second alloc instruction in a procedure does not cause overflow of stacked register as long as it resizes register stack frame to smaller one.

Douillet et al. first proposed a multi-alloc method to minimize the overhead associated with RSE spill/fill [5]. They analyze stack register usage per basic block and capture multiple control flows within a procedure that use different numbers of stacked registers. By inserting alloc multiple places in a procedure and resizes register stack frame differently for different paths, they could reduce the stacked register usage along some paths. Their scheme, however, does not show satisfactory results.

Settle et al. proposed another multi-alloc scheme that resizes register stack frame before call sites in order to reduce the total height of stack frames over all procedures in call stack [10]. The key idea is not-live registers across calls are overlapped with the callee’s register stack frame. Register liveness analysis is used to distinguish registers that can be overlapped. Although they provided many useful evidences that overlapping register stack frame would lead to RSE overhead reduction, their scheme takes advantage of only the not-live registers at the end of local-register region.

Weldon et al. examine various RSE optimizations techniques using dynamic register usage and dead register evaluation information due to imbalanced paths of applications’ execution [14]. They simulate heap-based RSE implementation, different from original stack-based RSE in Itanium.
architecture, in order to take advantage of those profile information fully, which assumes non-overlapping frames between callers and callees.

Yang et al. proposed a method that decides appropriate quota of stacked registers for each procedure [15]. Their approach is not based on multi-alloc method, but has the same goal to reduce the overhead of RSE spill/fill. They analyze call graph and use a cost model in order to find the trade-off between explicit register spills and usage of stacked registers managed by RSE. They observed that reducing stacked register usage in some procedures could reduce the total memory access time of spilling registers. Using their quota allocation method, perlbench, which originally spends 23% of total execution time in RSE, spends nearly 0% of cycles in RSE. They also reported 13% performance improvement for perlbench.

Our method presented in this paper follows the multi-alloc method used in the work of Settle et al. Our contributions over their work are as follows.

- We devise a register renaming scheme to maximize the overlap with callee’s register stack frame.

- We present a heuristic based on sequence graphs that can find optimal register renaming in consideration of multiple call sites within a procedure.

- We propose a profile-guided method to interprocedurally optimize the register renaming for register stack overflow minimization.

3 The Overflow-minimizing Register Renaming

To determine the size of the register stack frame, procedures often start with alloc instructions. Suppose procedure \( f() \) allocates 9 stacked registers, \( r32 \) through \( r40 \), ( in-registers: \( r32, r33 \), local-registers: \( r34 \) to \( r38 \), out-registers: \( r39, r40 \) ), as shown in Figure 2 (a) and (b). When \( f() \) calls \( g() \), out-registers of \( f() \) overlaps in-registers of \( g() \) as shown in Figure 2 (b). Figure 2 (c) illustrates multi-alloc method in [10]. Equipped with careful live register analysis, 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 two output registers \( r39 \) and \( r40 \) are renamed \( r37 \) and
Figure 2: Register stack height reduction: resizing of register stack frame and effect of register renaming.
r38 respectively. This resizing of f()’s register stack frame decreases register stack height for f() and g() by 2.

Now suppose f() calls another procedure h() besides g() as shown in Figure 2 (d). Still we can 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 rename some of the local-registers of f(), for example, 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 reduction on overall register stack height by 3 and 1 as shown in Figure 2 (f). This example clearly reveals the effect of register renaming on overall register stack height reduction, when call sites are more than one in a procedure. We can observe that, as long as the adjacent register on the right is live, it can not be eliminated from register stack frame even though it is dead. Pushing dead registers towards the end of local-register region is a key idea to reduce more of current register stack frame.

Since different call sites have different execution frequencies and some call sites lead to many stacked register overflows compared to others, we also exploit profiling information to establish priorities amongst call sites in a procedure. According to the calling convention, out-registers naturally overlap with callee’s register stack frame and in-registers convey parameters passed by callers of current procedures. The only candidates for safe renaming are local-registers. For those local-registers, all virtual register names are mutually interchangeable. Once we find one-to-one interchange mapping among local-registers, we can safely rename local-registers that result in more overlaps. The rest of this section discuss (1) how we define optimal register renaming, (2) how we convert the register renaming problem to graph problem, and (3) how we use profile to interprocedurally refine register renaming.

3.1 Optimal Register Renaming

For a procedure P, if there are k call sites C1, C2, · · · , Ck then the set of registers which are dead before the call site Ci (or unused until that point) is denoted as dead set Di. Let L = {r1, r2, · · · , rl} be the set of local-registers of the register stack frame for P. Now we want to find a sequence S, s1 → s2 → · · · → sl, si ∈ L, 1 ≤ i ≤ l, of all stacked registers in L, that
maximizes $G$:

$$G = \sum_{i=1}^{k} \text{gain}(S, D_i) = \sum_{i=1}^{k} \text{gain}(s_i, D_i) \tag{1}$$

$$\text{gain}(s_j, D_i) = \begin{cases} 1 + \text{gain}(s_{j-1}, D_i) & \text{if } s_j \in D_i \text{ and } 1 \leq j \leq l \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

When the original sequence of local-registers $S_0$ is $r_1 \rightarrow r_2 \rightarrow \cdots \rightarrow r_l$, we map the $i_{th}$ register in $S_0$ to the $i_{th}$ register in the new sequence $S$; then we find this bijective function $RN : L \rightarrow L$ as our register renaming. The gain of the register renaming $RN$ is $G$ given in Equation (1).

Equation (1) and Equation (2) represent the observation that 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 stack. We can stack up a new register frame on top of 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 $\text{gain}()$ is the number of registers that can be overlapped at each call site. The total gain, $G$, is a plain sum of all the gains at all call sites in the procedure $P$.

For example, suppose a procedure has two call sites and their dead sets are $D_1 = \{r_{46}, r_{45}, r_{48}\}$ and $D_2 = \{r_{43}, r_{45}, r_{48}\}$, and its set of all stacked registers in local region $L = \{r_{42}, r_{43}, \cdots, r_{48}\}$. Then the gain of a sequence, $S = r_{42} \rightarrow r_{44} \rightarrow r_{46} \rightarrow r_{47} \rightarrow r_{43} \rightarrow r_{45} \rightarrow r_{48}$, is computed to 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 ($\text{gain}(S, D_2) = 3$). Thus, the total gain is 5. One thing to note is that only the suffix of $S$ which consists of dead registers matters. The order of the rest registers are irrelevant to the total gain.

### 3.2 Optimal Register Renaming Using Sequence Graphs

Our register renaming is more easily tackled by relating it to a graph optimization problem as shown in Figure 3. Converting the original problem into a graph problem gives more intuitions on how to solve our problem. To find a register renaming that produce maximum gain, we attempt 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 the sequence, we try all possible permutations of dead registers in all dead sets of all call sites. That means we attempt to find a permutation for each dead set which results in maximum total gain.

Now we explain how we convert our problem to equivalent graph problem using an example. Figure 3 (a) shows two dead sets $D_1$ and $D_2$ from our previous example, where a function $f()$ calls $g()$ and $h()$ at two call sites $C_1$ and $C_2$, respectively. In Figure 3 (b), complete graphs for dead set $D_1$ and $D_2$ are shown. Any possible sequence of registers of each dead set can be represented as a path on each complete graph. The superimposition of two complete graphs is shown in (c), where edges from complete graph $D_1$ are denoted by dotted lines and those from $D_2$ are denoted by solid lines. The weights of nodes and edges are one for now. When we compose a superimposed graph, the weight of each node and edge is the sum of weights from all original graphs.

Suppose we pick a sequence of nodes that ends with a subsequence, $r43 \to r45 \to r48$. In $D_1$, only $r45$ and $r48$ can be overlapped from local part of register stack frame, since $r46$ does not appear consecutively with other dead registers. Whereas, $r43$, $r45$ and $r48$ in $D_2$ can be all overlapped, since all dead registers consecutively appear at the end of the sequence. As a result, we could reduce register stack frame by 2 and 3 before call site $C_1$ and $C_2$, respectively. On the complete graph of $D_1$, we decide to overlap a path $r48 \to r45$. On the complete graph of $D_2$, we decide to overlap a path $r48 \to r45 \to r43$. Projecting them to the superimposed graph in (c) and adding the weights of the first node and the following edges, we can get the total node weight of overlapped path ($5 = 2 + 2 + 1$), which is the same as the total gain of the overlapped register stack frame.

From the above example, we infer that we can calculate the gain of a register renaming using corresponding sequence on superimposed graph. The superimposed graph as shown in Figure 3 (c) is a multi-graph which has multiple edges between two nodes. To simplify the graph, we construct a sequence graph $G(V, E)$, where $V = D_1 \cup D_2 \cup \cdots \cup D_k$, and $E = \{(u, v) : u \in D_i$ and $v \in D_j\}$. The weight of a node, $w(u)$, is the sum of weights from all complete graphs of dead sets it belongs to, where weights of complete graphs are determined by the weights of their associated call sites. An edge $(u, v)$ has two attributes, weight and annotation. The weight of an edge, $w(u, v)$, is also the sum of weights from all complete graphs of dead sets it belongs to.
The annotation of an edge, \( a(u, v) \), is set of all dead sets to whose complete graphs it belongs.

Figure 3 (d) shows an example of sequence graph composed from dead sets listed in Figure 3 (a). On sequence graphs, we need to find a node-sequence which result in maximum gain, where we formally define node-sequence as follows.

- Let a sequence of nodes \( NS = v_1 \to v_2 \to \cdots \to v_q \), \( q \leq |V| \) be node-sequence from \( G(V, E) \). \( NS \) satisfies the following conditions.

  1. \( \forall 1 \leq i \leq q - 1, \text{ edge } (v_i, v_{i+1}) \in E \) and,

  2. \( \forall 2 \leq i \leq q - 1, a(v_i, v_{i+1}) \subset a(v_{i-1}, v_i) \) and,

  3. The gain of \( NS \) is the sum of the node weight of the first node in \( NS \) and all following edges’ weights in \( NS \).

\[
\text{gain of } NS = w(v_1) + \sum_{i=1}^{q-1} w(v_i, v_{i+1}) \quad (3)
\]

The first condition is a trivial condition, which enforces the sequence forms a path on \( G(V, E) \). The second condition implies that we remember the dead sets we are overlapping and select the next edge among the complete graphs of dead sets that can extend the overlap. The third condition defines the gain of a sequence. To find optimal node-sequence, we pick 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 node-sequence of dead registers, we reverse it as we select the rightmost dead register first for the node-sequence. We then arbitrarily compose a sequence holding the rest of the local registers.
We finally concatenate two sequences, putting the sequence of dead registers in the back. The resulting sequence can now be used to guide the register renaming. In section 4 we will describe how to find maximum gain node-sequence from sequence graphs using modified spanning tree algorithm. Formal proof of equivalency between optimal register renaming and maximum gain node-sequence is also given in Appendix A.

3.3 Profile-guided Register Renaming

When we attempt to find overlapping registers within dead sets, we have to overlap with as many dead sets as possible. However, we always need a decision policy which dead sets we overlap. Relative importance of each call site can be a good guidance. To differentiate the multiple call sites within a procedure, we profile the frequencies of calls and keep track of the chains of call sites which lead to register stack overflow. To accomplish the profiling, we first run the application and produce dynamic call site list. At each call site we collect caller’s address and callee’s address. The GNU compiler allows to insert instrumentation code at function entries and exits. With the instrumentation code and customized linkage editor option, we can collect the information about dynamic call chains. Using GNU *binutils* and instrumented executable files generated by GNU compiler, we translate addresses of call sites to static source-level information such as caller’s name, callee’s name and position at source code. By inserting instrumentation codes in ORC for Itanium processors [7], we can collect the configuration of register stack frame of each procedure. Now we have enough information to simulate the register stack engine. Simulating the call chains and register stack, we can have dynamic call site graphs. An example of call site chains is shown in Figure 4 (a). During the simulation, we mark the overflow points. When a overflow occurs, we track down the call stack to find the recent overflow point. We can then compose a overflow call chain by listing all the dynamic call sites in stack from the recent overflow to the current overflow. Figure 4 (b) shows overflow call chains collected from the example call site graph in Figure 4 (a). After finding all the overflow call chains, we count the number of occurrences in the overflow call chains for each call site. The frequency becomes the weight of the call site. In previous section, we initially set to one for the weights of nodes and edges in the complete graphs for dead sets. Now, we set the weight of each call site to the invocation frequencies that are directly connected to register overflows, as described in this section. Using profile-based weights, we can get better
register renaming to minimize register stack overflows.

4 Algorithm

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

4.1 rename_register and multi_alloc

Our proposed algorithm, called rename_register, is based on a greedy approach similar to Prim’s maximum spanning tree algorithm. The input to rename_register is the dead sets \((D_1, D_2, \ldots, D_k)\) of a procedure. From \(D_1, D_2, \ldots, D_k\), we build a corresponding sequence graph \(G\). We then construct a node-sequence by appending one edge at a time. Our algorithm starts with a node-sequence \(NS\) that contains only one node \(u\) whose weight is the biggest among all other nodes in \(G\). Within the while body, node-sequence appends a new edge which has the biggest weight. However, we cannot add any edges to node-sequence. In order to get in to the node-sequence, the edge needs to satisfy a certain condition described on line 9 of Algorithm 1. The condition implies that we can only expand node-sequence by adding an edge which belongs to the dead sets annotated on the last edge of the node-sequence. That means the algorithm attempts to overlap the node-sequence with multiple dead sets so far. Thus, we can extend the overlap only with
Algorithm 1 rename_register()

Input : Dead sets, \(D_1, D_2, \ldots, D_k\), for all call sites in a procedure \(P\).

Output: A sequence of nodes.

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

those dead sets, but not with new dead sets. Algorithm 1 contains a sketch of \texttt{rename_register}.

Figure 5 shows an example of \texttt{rename_register} step by step. In Figure 5 (a), there are 4 distinct
dead sets. The registers in each dead set are given simple names such as \(a, b, \ldots, f\), for ease
of description. The number after each dead set indicates the weight of each dead set. The
weight is obtained from the profiling data for corresponding call site. Figure 5(b) shows the
superimposition of four complete graphs from \(D_1, D_2, D_3\) and \(D_4\), while Figure 5(c) shows the
resulting sequence graph. Figure 5(d) through (g) illustrate each step how we find the maximum
gain node-sequence. At first, node \(a\) is chosen, since it has the biggest weight. Edge \((a, e)\) is
chosen since it has the biggest weight among edges starting from \(a\). We pick edge \((e, b)\) as next
dge among \((e, b), (e, c), (e, d)\). Since the end node of current sequence is \(e\), we first select all edges
from \(e\). We then check the annotations of those edges \((a(e, b) = a(e, c) = D_2), a(e, d) = D_1\)\).
All the annotations are subset of \(a(a(e), = \{D_1, D_2\})\). Thus, we pick an edge with the biggest
weight, which is \((e, b)\) (actually, \(w(e, b) = w(e, c), (e, b)\) is chosen arbitrarily between the two.)
With a similar step, \((b, c)\) is picked at the next iteration. Now, there is no edge adjacent to \(c\) that
satisfies the condition in line 9 of Algorithm 1. Our algorithm stops here, resulting a sequence
\(a \rightarrow e \rightarrow b \rightarrow c\). The gain from this sequence is \(24 = 9 + 7 + 4 + 4\).
Figure 5: An example illustrating each stage of edge addition to build a node sequence.
Algorithm 2 multi_alloc()

Input: Optimized code of a procedure \( P \) after register allocation, Weights of call sites from profile

Output: Register stack optimized code with multiple \texttt{alloc} instructions.

1: Let \( D \) be a set of all dead sets in \( P \)
2: for all call site \( C \) in \( P \) do
3: Calculate dead set \( D \) for call site \( C \) /* analyze dead registers before this call site. */
4: \( D = D \cup \{ D \} \)
5: \( w(D) = w(C) \) /* \( w(C) \) denotes the weight of call site \( C \) obtained from profile */.
6: end for
7: \( \hat{NS} = \text{rename_register}(D) \) /* \( \hat{NS} \) is a reversed node-sequence */
8: Let \( L \) be set of local registers of \( P \).
9: \( S \leftarrow \) arbitrarily concatenate all nodes in \( L - V(\hat{NS}) \) before \( \hat{NS} \).
10: Rename local registers used in \( P \) using \( S \).
11: for all call site \( C \) in \( P \) do
12: Let \( D \) be the dead set of call site \( C \)
13: if decide to resize based on \( \hat{NS}, D \) then
14: Resize register stack before and after \( C \) using \( \hat{NS}, D \). /* insert alloc and copy parameters */
15: end if
16: end for

Note that we can come up with simpler heuristic than \texttt{rename_register}(), in which we only consider how many times a register appears in dead sets (namely, node frequency). We sort nodes in all dead sets in non-decreasing order of node frequencies, then the result will produce a node-sequence. With this heuristic the example in Figure 5 results in \( a \rightarrow e \rightarrow c \rightarrow d \rightarrow b \rightarrow f \). The gain of this node-sequence is 20 (= 9 + 7 + 4), which is less than 24, the gain we can get from \texttt{rename_register}(). This simple heuristic has some advantages over \texttt{rename_register}(). It does not build sequence graph, but just counts the node frequency of each register. As a result, it has lower time complexity than that of \texttt{rename_register}(), but with less optimized register renaming.

Algorithm 2 shows how we use multi-alloc method. Optimized code of a procedure after register allocation is given as an input. The algorithm also takes weights of call sites, which are calculated from profile information. First we gather all the dead sets in a procedure, assigning a weight to each dead set. The weight of a dead set becomes the weights of nodes and edges in the dead set. With weighted dead sets, we obtain register renaming for local registers. After we interchange local register names with the register renaming, we insert multiple \texttt{alloc}'s wherever they are
necessary. The alloc’s inserted to resize register stack frame also require to copy parameters from old out-registers to 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, set operations on annotations are able to run in constant time using bitwise operations. Since we construct complete graph for each dead set, the complexity of building sequence graph is \(O(|D_1|^2 + |D_2|^2 + \cdots + |D_k|^2) = O(\sum_{i=1}^{k} |D_i|^2)\). When we iterate while loop and expand the node-sequence, all adjacent edges to the last node of the sequence need to be examined. The upper bound for any node’s degree is \(\frac{|D_{\text{max}}|^2|D_{\text{max}}-1|}{2}\), where \(D_{\text{max}}\) is the dead set of maximum cardinality among \(D_1, D_2, \cdots, D_k\). Thus, the complexity of while loop is \(|V| \cdot \frac{|D_{\text{max}}|^2|D_{\text{max}}-1|}{2}\). Overall, the complexity of rename_register is bound by \(\max(\sum_{i=1}^{k} |D_i|^2, |V(G)| \cdot \frac{|D_{\text{max}}|^2|D_{\text{max}}-1|}{2})\).

5 Experimental Results

We perform our experiments on an Intel Itanium2 box with two 1.4GHz Itanium2 processors and 1.5M L2 cache, running 64bit version of RedHat Linux. To collect simulation data, we implement our register_renaming and multi_alloc algorithms on the Open Research Compiler(ORC). For various simulation data collection, we also use GNU compilers for Itanium architectures. SPEC CINT2000 benchmarks are used to evaluate the effectiveness of our approach. Even though we generate multi-alloc codes for the benchmark programs, we could not run those codes on our Itanium box. Itanium architectures are allowed to use multi-alloc on Software Developer’s Manual [6], but commercial boxes do not seem to turn on that feature. Thus, the results shown in our paper are based on our own simulation of register stack.

5.1 Characteristics of Benchmarks

Our register_renaming is applied after register allocation phase of ORC. The register allocator in ORC follows the principles in Chow’s allocator [4, 3] and Chaitin-Briggs’ allocator [2, 1].
This register allocator classifies stacked registers into stacked callee-save and stacked caller-save. The former is used for live registers across call sites and allocated in in/local-register regions. Meanwhile, the latter is mainly for 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. The technique we use further finds dead registers in local-register region.

Table 1 lists static characteristics of SPEC CINT2000 programs. Each column out of nine shows: (1) names of benchmarks, (2) numbers of total call sites, (3) average numbers of call sites per procedure, (4) average percentages of call sites per procedure where our register stack frame resizing algorithm is applied, (5) average numbers of stacked registers allocated to a procedure (in/local/out-registers), (6) average numbers of stacked registers which are dead before at least one call site, (7) average numbers of local-registers which are dead before at least one call site, (8) average numbers of local-registers which are overlapped with callee’s register stack frame only using multi-alloc, and (9) average numbers of local-registers which are overlapped with callee’s register stack frame using multi-alloc and register renaming.

Referring to three columns for call sites, we found SPEC CINT2000 benchmarks have about 9 static call sites in a procedure on average. Among those call sites, about 40% of them resize default register stack frames. One extreme case is crafty, which has about 30 call sites in a procedure. This is about three times more than average. Our multi-alloc scheme with register renaming resizes register stack frame for more than half of those call sites.

Referring to five columns for stacked registers, we collected the information from the codes generated by ORC. Figure 6 also graphically shows these characteristics for each benchmark. Total number of stacked registers are about 12, which is far less than allowed number of registers, 96. Among stacked registers, we found about 6 registers are dead at call sites. Although half of stacked registers are dead at call sites, 4–5 registers out 6 are out-registers, which are already overlapped with callee’s register stack frame. Only 1–2 registers are dead among local-registers, which are about 16% of in/local-registers. Since register stack is a pile of in/local-registers of previous procedures in call stack, 16% is our upper bound in register stack reduction. Naive multi-alloc scheme overlaps only a third of dead local-registers. Our sophisticated multi-alloc scheme, which exploits register renaming, can overlap 90% of dead local-registers.
Figure 6: Static numbers of registers per procedure for SPEC CINT2000.
5.2 Reduction in Overflowed Registers

Using simulation of register stack, we collect the number of overflowed registers. Dynamic call chains collected from GNU compilers and source line matches with GNU *binutils* are provided to our register stack simulator. Static information about default configurations and resizes at call sites for register stack frame collected using ORC are also provided to our simulator. Using the provided dynamic and static information, our simulator produces numbers of overflowed registers for SPEC CINT2000 benchmarks.

Table 2 shows the numbers of overflowed registers during the simulation of benchmarks. We compared three techniques to codes generated by ORC compiler (*base*). In multi-alloc, *alloc* instructions are added before call sites and attempt to overlap dead local-registers at the end of local-register regions as described in [10]. The other two are our multi-alloc schemes with register_renaming and profiling information. Figure 7 shows the percentage of reduction over *base* in numbers of overflowed registers from register stack simulation.

Gcc shows the highest reduction of 27%. Crafty, parser, perlmbk, and gap also have reduction more than 10% over *base*. Our profile-guided register renaming results in reduction of 7.1% in overflowed registers on average. With respect to naive multi-alloc, multi-alloc with register_renaming achieves additional reduction of 3% on average. As for gcc and gap, register...

---

**Table 1**: Static characteristics of SPEC CINT2000.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>in program</th>
<th>per procedure</th>
<th>alloc-site percentage</th>
<th>total</th>
<th>dead regs. (all)</th>
<th>dead regs. (local)</th>
<th>overlapped local-reg.</th>
<th>no rename</th>
<th>with rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>254</td>
<td>4.98</td>
<td>40%</td>
<td>10.00</td>
<td>4.36</td>
<td>1.33</td>
<td>0.38</td>
<td>1.20</td>
<td></td>
</tr>
<tr>
<td>vpr</td>
<td>1514</td>
<td>8.96</td>
<td>47%</td>
<td>14.32</td>
<td>6.78</td>
<td>1.54</td>
<td>0.38</td>
<td>1.39</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>17822</td>
<td>10.80</td>
<td>40%</td>
<td>10.87</td>
<td>5.53</td>
<td>1.34</td>
<td>0.56</td>
<td>1.21</td>
<td></td>
</tr>
<tr>
<td>mcf</td>
<td>54</td>
<td>6.75</td>
<td>36%</td>
<td>11.13</td>
<td>5.13</td>
<td>1.66</td>
<td>0.00</td>
<td>1.56</td>
<td></td>
</tr>
<tr>
<td>crafty</td>
<td>405</td>
<td>29.42</td>
<td>55%</td>
<td>15.54</td>
<td>8.42</td>
<td>2.13</td>
<td>1.10</td>
<td>1.81</td>
<td></td>
</tr>
<tr>
<td>parser</td>
<td>1214</td>
<td>5.62</td>
<td>30%</td>
<td>9.26</td>
<td>3.57</td>
<td>0.85</td>
<td>0.21</td>
<td>0.72</td>
<td></td>
</tr>
<tr>
<td>eon</td>
<td>7422</td>
<td>3.15</td>
<td>12%</td>
<td>7.83</td>
<td>3.78</td>
<td>0.23</td>
<td>0.07</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>perlmbk</td>
<td>5775</td>
<td>6.72</td>
<td>40%</td>
<td>10.64</td>
<td>5.40</td>
<td>1.32</td>
<td>0.54</td>
<td>1.13</td>
<td></td>
</tr>
<tr>
<td>gap</td>
<td>3576</td>
<td>6.62</td>
<td>47%</td>
<td>11.21</td>
<td>5.67</td>
<td>1.75</td>
<td>0.63</td>
<td>1.55</td>
<td></td>
</tr>
<tr>
<td>vortex</td>
<td>8097</td>
<td>9.32</td>
<td>37%</td>
<td>16.22</td>
<td>9.39</td>
<td>0.90</td>
<td>0.54</td>
<td>0.86</td>
<td></td>
</tr>
<tr>
<td>bzip2</td>
<td>254</td>
<td>5.52</td>
<td>39%</td>
<td>8.63</td>
<td>4.47</td>
<td>0.95</td>
<td>0.23</td>
<td>0.89</td>
<td></td>
</tr>
<tr>
<td>twolf</td>
<td>611</td>
<td>8.04</td>
<td>45%</td>
<td>15.67</td>
<td>6.88</td>
<td>2.10</td>
<td>0.56</td>
<td>1.83</td>
<td></td>
</tr>
<tr>
<td>Avg.</td>
<td>3615</td>
<td>8.82</td>
<td>39%</td>
<td>11.78</td>
<td>5.78</td>
<td>1.34</td>
<td>0.43</td>
<td>1.20</td>
<td></td>
</tr>
</tbody>
</table>
renaming adds reduction of more than 10%. As we discussed in the previous section, register renaming enables us to overlap three time more dead registers than naive multi-alloc. As a result, we get more reduction in overflowed registers by using register renaming. The benchmarks which show reductions using register renaming achieve about 2% more reductions by using profile information. One exception is `crafty`, which results in 7% more reduction with profiling. As pointed out earlier, `crafty` has many call sites within a procedure. If we equally treat all the call sites, our register renaming algorithm arbitrary favors one call site among available choices at each decision point. By providing profile information, we can guide our register renaming algorithm to favor important call sites which need more overlap than others. The more call sites we have in a procedure, the more effective giving a guidance from profile is.

### 6 Conclusion

Hardware managed register stack 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 limited number of virtual registers due to encoding space on instruction set architectures. To better support hardware managed register stack, we apply compiler-assisted register renaming.

---

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>base</th>
<th>multi-alloc</th>
<th>multi-alloc + register renaming</th>
<th>multi-alloc + profile-guided register renaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>17714</td>
<td>17694</td>
<td>17058</td>
<td>17058</td>
</tr>
<tr>
<td>vpr</td>
<td>5269</td>
<td>5267</td>
<td>5263</td>
<td>5263</td>
</tr>
<tr>
<td>gcc</td>
<td>81381080</td>
<td>72237243</td>
<td>60192948</td>
<td>59651099</td>
</tr>
<tr>
<td>mcf</td>
<td>512883</td>
<td>512883</td>
<td>512881</td>
<td>512881</td>
</tr>
<tr>
<td>crafty</td>
<td>109854035</td>
<td>107304044</td>
<td>102530307</td>
<td>95055454</td>
</tr>
<tr>
<td>parser</td>
<td>29079149</td>
<td>26430493</td>
<td>25101873</td>
<td>24818246</td>
</tr>
<tr>
<td>eon</td>
<td>3689578</td>
<td>3689578</td>
<td>3689566</td>
<td>3689566</td>
</tr>
<tr>
<td>perlbmk</td>
<td>1015422</td>
<td>934529</td>
<td>906058</td>
<td>901255</td>
</tr>
<tr>
<td>gap</td>
<td>15988097</td>
<td>15701895</td>
<td>13797744</td>
<td>13555430</td>
</tr>
<tr>
<td>vortex</td>
<td>42879780</td>
<td>42879780*</td>
<td>42879780*</td>
<td>42879780*</td>
</tr>
<tr>
<td>bzip2</td>
<td>4692744</td>
<td>4692744</td>
<td>4692744</td>
<td>4692744</td>
</tr>
<tr>
<td>twolf</td>
<td>3868922</td>
<td>3868921</td>
<td>3868920</td>
<td>3868920</td>
</tr>
</tbody>
</table>

| Avg. reduction | 0.00% | 3.0% | 6.2% | 7.1% |

Table 2: Dynamic numbers of overflowed registers in register stack.
Figure 7: Reductions in overflowed registers using multi-alloc and our register renaming algorithms.

Given that multiple alloc instructions are supported in a procedure, we can also freely resize the register stack frame multiple times within a procedure. Using both register renaming and resizing register stack frame, we achieve maximal overlapping on procedure calls. Since our register renaming approach is applied after 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.

While previous works such as in [10] try to overlap only the local registers adjacent to out-registers, our approach overlaps almost all local registers that are dead (not-live). The key component of our scheme is to find optimal register renaming which enables us to overlap those dead local registers. According to our experiment with SPEC CINT2000, while multi-alloc alone reduces overflowed registers in RSE by 3.0%, multi-alloc with register renaming reduces by 6.2%. Furthermore, we use profile of call site chains and register stack overflows to distinguish relative importance of each call site. When we use profile, our register renaming algorithm finds more optimized register renaming that considers overflow call chains. Our experiment tells multi-alloc with profile-guided register renaming reduces overflowed registers by 7.1% on average. Five benchmarks (gcc, crafty,
parser, perlbmk, and gap), however, show large reductions by 12–27%.

Since the overhead of register stack engine is less than 2% of total execution time for many applications in SPEC CINT2000 [15], our scheme might not contribute to the performance for many applications. Some of SPEC CINT2000 such as perlbmk and crafty, however, suffer from the RSE overhead that mounts up to 24%. We also expect many complex applications written in object-oriented languages or functional languages suffer from RSE overhead due to deep call stack from many calls of methods and functions. For those applications, our technique will be still useful to increase the performance.

References


A Equivalence of overflow-minimizing register renaming and maximum gain sequence finding

We claim that the gain of a node-sequence of sequence graph is the same as the gain of the register renaming. Let $D = D_1 \cup D_2 \cup \cdots \cup D_k$ and $L$ be the set of all stack registers in local region. Then problem instance of overflow-minimizing register renaming problem is denoted as $(D, L, RN)$, where $RN$ is a bijective register renaming mapping from $L$ to $L$. On the other hand, $(D, L, NS)$ denotes a problem instance of maximum gain node-sequence finding problem in sequence graph of $D_1, D_2, \cdots, D_k$.

Given an $RN$ of $(D, L, RN)$ there is a sequence $S$ of all stacked registers in $L$ by definition of the overflow-minimizing register renaming problem. Let $\tilde{S}$ be the reverse sequence of $S$. Then implied $RN$ of $RN$ is the longest prefix of $\tilde{S}$, where all registers of the prefix are in $D$. Then we have following theorem.

**Theorem 1** Given $(D, L, RN)$ and $(D, L, NS)$, if $NS$ is implied from $RN$, the gain $G$ of $RN$ is the same as the gain of $NS$.

**Proof 1** Let $S$, $s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_l$, be the sequence of all registers in $L$ determined by $RN$. Then implied $NS$ is the longest subsequence of $\tilde{S}$, $s_1 \rightarrow s_{l-1} \rightarrow \cdots \rightarrow s_q$ ($1 \leq q \leq l$), where $\forall q \leq m \leq l$, $s_m \in D$. Suppose the gain $G$ of $RN$ is less than the gain of $NS$. Then by definition of sequence graph and gain of a node-sequence, the difference results either from (1) node weight of $s_l$ or from (2) weight of an edge $(s_m,s_{m-1})$, $(q+1 \leq m \leq l)$ of $NS$, if the edge exists in sequence graph of $D$. In the case of (1), some occurrences of $s_l$ in some dead sets were not added to $G$ of $RN$, which contradicts to the definition of $G$ in Equation (1). Let $w$ denotes weight of edge, $(s_m,s_{m-1})$. In the case of (2), $s_{m-1}$ contributes less than $w$ to $G$ though all of $s_1$, $s_{l-1}$, $s_m$ are in the same dead sets $w$ times together, which also contradicts to the definition of $G$.

Assuming $G$ of $RN$ is more than the gain of $NS$, there are contradictions in similar way. Thus, $G$ of $RN = gain$ of $NS$.

Conversely, implied $RN$ of $NS$ in $(D, L, NS)$ is defined as following: 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 reversion of the resulting sequence; implied $RN$ is determined from $S$ by definition of the overflow-minimizing register renaming problem.
Theorem 2  Given $(D, L, RN)$ and $(D, L, NS)$, if $RN$ is implied from $NS$, the gain of $NS$ is the same as the gain $G$ of $RN$.

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