Fine-Grain Stacked Register Allocation for the Itanium Architecture

Alban Douillet **José Nelson Amaral** **Guang R. Gao**

Dept. of Computer Science
University of Delaware
Newark, DE 19716
U.S.A.

Dept. of Computing Sciences
University of Alberta
Edmonton, Alberta, T6G 2E8
Canada

Dept. of Electrical Engineering
University of Delaware
Newark, DE 19716
USA

{douillet,ggao}@capsl.udel.edu, amaral@cs.ualberta.ca

Abstract. The introduction of a hardware managed register stack in the Itanium Architecture creates an opportunity to optimize both the frequency in which a compiler requests allocation of registers from this stack and the number of registers requested. The Itanium Architecture specifies the implementation of a Register Stack Engine (RSE) that automatically performs register spills and fills. However, if the compiler requests too many registers, through the *alloc* instruction, the RSE will be forced to execute unnecessary spill and fill operations. In this paper we introduce the formulation of the fine-grain register stack frame sizing problem. The *normal* interaction between the compiler and the RSE suggested by the Itanium Architecture designers is for the compiler to request the maximum number of registers required by a procedure at the procedure invocation. Our new problem formulation allows for more conservative stack register allocation because it acknowledges that the number of registers required in different control flow paths varies significantly. We introduce a basic algorithm to solve the stack register allocation problem, and present our preliminary performance results from the implementation of our algorithm in the Open64 compiler.

1 Introduction

The problem of minimizing data traffic between the memory and the registers of a processor — known as the register allocation problem — has occupied researchers for many years. Whether selecting a set of values to be *promoted* to registers [5], or minimizing the number of values spilled from registers to memory [4, 3], the goal is to minimize the number of loads and stores actually executed at runtime to reduce memory traffic and thus reduce the execution time of the program.

Register allocation algorithms work with the constraint that a processor has a fixed — and often small — register set. Besides the increased traffic with memory caused by the unavoidable spill operations, reusing the same register to store multiple temporary values introduces write after read (WAR), and write after write (WAW) dependencies in the instruction stream. Such dependences are not intrinsic to the program being executed, but are a consequence of register reuse.

Another unintended consequence of the small fixed-size register file is that the load and store instructions required for register spilling must be fetched from memory and issued, thus these instructions compete with other instructions for space in the instruction and the data cache and further increase the memory traffic.

In order to eliminate these avoidable dependences, out-of-order issue processors often have extra *non-architected* registers — or *reservation stations* — that are not visible to the compiler. This extra storage can be used at runtime to rename the registers selected by the compiler, eliminating the extra dependences and allowing more instruction level parallelism. Unfortunately loads and stores inserted by the compiler to spill values to memory cannot be eliminated from the instruction stream at runtime. Therefore these spill instructions increase the memory traffic even when some of the non-architected storage could be used to save the value been spilled[9, 10].

An alternative design that eliminates many of these problems is adopted in the Intel Itanium Architecture [6–8]. In the Itanium a portion of the register file is implemented as a very deep stack. In the first processor in the Itanium family, the top 96 positions of this register stack are implemented as physical registers, while the remainder of the stack is mapped to memory. An instruction, called *alloc* , is provided to enable the compiler to specify how many registers will be used by each procedure. This instruction allows for up to 96 registers to be allocated at once. The architecture also provides a *register stack engine* (RSE), a hardware mechanism that automatically copies to and from memory the bottom portion of

the stack that does not fit in the 96 physical registers. To the best of our knowledge, the Itanium architecture is the only architecture that uses such a mechanism.

Whenever the accumulated allocations in a program exceed 96 registers, the RSE transfers values between the memory and the registers to make room for the new allocation. Therefore the compiler still has to solve the register allocation problem in a similar fashion as it does for architectures without a register stack. However it is now possible to make new tradeoffs between serialization caused by the creation of WAR and WAW dependences and the allocation of more registers. Moreover the allocation instruction itself has a cost that needs to be taken into consideration when multiple allocation instructions are used in a procedure to reduce the accumulated register allocation.

The *alloc* instruction was designed to be called once at the beginning of every function. In this paper, we propose the *fine-grain allocation of stacked registers*, *i.e.*,we propose to use more than one *alloc* instruction in each procedure in order to reduce the number of unnecessary register spills and fills. In Section 2, we describe the register stack and the *alloc* instruction. In Section 3, we introduce a motivating example and clearly formulate the multi-*alloc* problem. In Section 4, we describe an algorithm to solve the multi-*alloc* problem. The experimental results are presented in Section 5 and show that a finer-grain use of the *alloc* instruction can lead to improvements at run-time.

2 Register Stack & Allocation Instruction

The Itanium architecture has 128 integer general purpose registers. Of those, 32 are static registers accessed and allocated by the compiler using conventional mechanisms. A *register stack* is implemented in the remaining 96 registers. Because the architecture maintains a backing storage where portions of the stack can be spilled, from the point of view of the application, this stack can grow unbounded. Stacked registers are organized into *frames*, one per function invocation. The size of the frames are set using the *alloc* instructions[1]. Each individual alloc instruction can resize the current register stack frame to up to 96 registers.

Whenever the total number of stacked registers allocated surpasses 96, a hardware mechanism, called the *Register Stack Engine* (RSE) automatically spills enough values to the backing storage to make room for a new allocation request. When physical registers become available (e.g. due to the completion of a function invocation, the RSE fills these registers with values that had been previously spilled. The spill/fill operations are asynchronous with the execution of the instructions of the running application.

2.1 The Allocation Instruction

The alloc instruction has four parameters: the number of inputs, i, the number of locals, l, the number of outputs, o, and the number of rotating registers, r. The size of the frame allocated is given by $l + o$. The input registers are a subset of the local registers. The output registers of a caller procedure overlap with the input registers of the callee to allow the passage of parameters via registers. The rotating registers are a subset of the stacked registers allocated in the current frame with the restriction that $0 \leq r \leq l + o$. Rotating registers are used to enable the implementation of dynamic single assignment in software pipelined loops. The execution of an alloc instruction may either grow or shrink the register frame of the current procedure. The parameters of the alloc instruction specify the size of the current frame, and that this new size is effective immediately upon completion of the instruction.

For simplicity, in the remaining of this paper we consider the *alloc* instruction to have a single parameter that is the size of the frame. Unless otherwise stated, henceforth, all references to number of registers, refer to stacked registers. We say that a function requires n registers for its execution if in at least one of its execution paths n stacked registers are accessed. Notice that not all the executions of the function will need the allocation of n registers, as the function might not execute the path that requires the maximum number of registers. Consider, for instance, a function foo that requires 60 registers and that calls a function bar that also requires 60 registers. Figure $1(a)$ shows the register stack after the allocation of registers for the function foo. Figure $1(b)$ illustrates that when bar executes there is not enough registers to allocate its 60 registers (see the shaded area). Therefore some of the registers previously allocated to foo must be saved to memory (spilled) to make room for the registers required by bar (Figure $1(c)$). The register frame of bar wraps around to use the space emptied by spilling the lower part of foo's frame. Now consider that we have used our technique in bar and have provided multiple *alloc* instructions for different paths of bar. If the current invocation of bar only requires 10 registers, the pattern

[1] The *alloc* instruction should actually be named *resize* instruction. Indeed it does not only allocate registers but also deallocate them if needed. The effect of the instruction is only a change of size of the register stack frame

of allocation will be the one shown in Figure 1(d), and no register spilling by the RSE would be required.

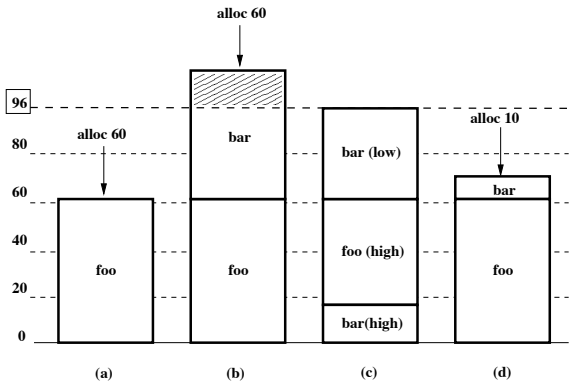

Fig. 1. Example of the Effects of the *alloc* Instruction

In this paper we explore the use of multiple alloc instructions in a procedure in order to reduce the number of unnecessary spills/fills performed by the RSE. Our intuition is that if the compiler is forced to use a single alloc instruction per procedure, this instruction must inserted early in the procedure and must request the allocation of the maximum number of registers used in any control path through the procedure. If the total number of allocated registers over all the active functions exceeds 96, then the RSE must spilled values in all called procedures. Meanwhile the actual register requirement in some control paths may be considerably smaller than the maximum among all control paths.

2.2 RSE Modes of Operation

An important factor in the optimization of the placement of *alloc* instructions by the compiler is the policy used to perform the spill and fill operations by the RSE. The Itanium architecture proposes four spill/fill policies for the RSE implemented as modes of operation. The four modes of operations offer combinations of *eager* and *just-in-time* loads and stores. A load/store is said to be just-in-time when it is executed when an *alloc* instruction triggers it or by the return of a procedure. A load/store is said to be eager when the RSE speculatively loads/stores registers from/to memory before an *alloc* instruction asks for it or the procedure returns. Through the eager execution of load/stores, the RSE will hopefully make enough space for the next *alloc* instruction and will not stall the execution of the program waiting for the spills to be executed.

Although the algorithm discussed in this paper is independent of the mode of operation of the RSE. the "eager loads/eager stores" mode of operation would be the most efficient one for applications with many function calls. However, the Itanium processor only implements the "just-in-time loads/just-in-time stores".

3 Problem Statement and Motivating Example

In this section we introduce a simple example that we will used throughout the paper to motivate and describe the execution of our multi-*alloc* algorithm.

3.1 Motivating Example

We consider the problem of efficiently inserting *alloc* instructions in the code of a function f in a program P. We explain what we mean by "efficient" through the following example. We are given the Control-Flow Graph (CFG) G of f and the *local register requirement* (lrr) of every basic block of G, *i.e.*,the number of *stacked* registers that must be allocated for each basic block of G. For instance, $lrr(A) = 10$ means that basic block A requires that at least 10 registers be allocated from the register stack to execute properly. We assume that the CFG is acyclic — we will deal with loops later. Also, the lrr values are known and our problem formulation takes place after the register assignment phase[2].

Figure 2 shows the CFG G that we will use throughout the paper. The big boxes represents the basic blocks of G while the number in the little boxes attached to the basic blocks are the lrr value of the corresponding basic block. For instance, $lrr(C) = 20$ and $lrr(F) = 96$.

[2] Usually named *register allocation phase*, but we want to avoid any confusion with the traditional register allocation problem.

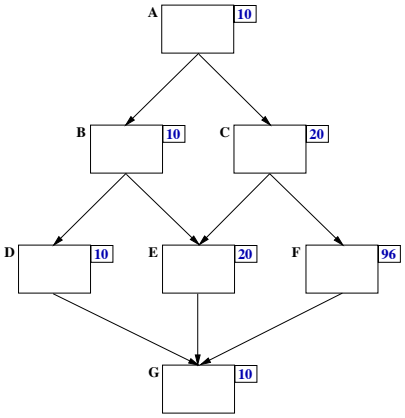

Fig. 2. the CFG G of a routine f where every basic block is associated with its lrr value.

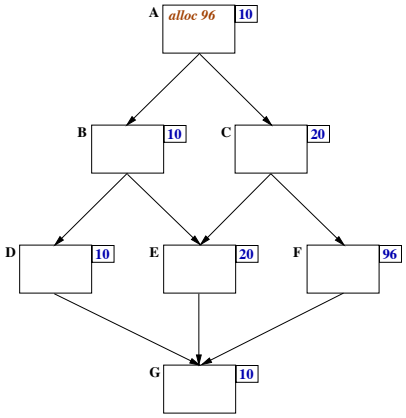

Fig. 3. The *alloc* instruction allocates enough registers so that every basic block in G can execute properly

The *alloc* instructions have not been inserted in the code of f yet. For instance, the basic block A can only be executed if there are at least $lrr(A) = 10$ registers allocated on the stack. Thus, we must make sure that,

Criterion 1: For every control-flow path C of G, there will be enough registers allocated to allow the execution of every basic block of C .

Figure 3 presents an allocation instruction insertion scheme that satisfies *Criterion 1*

Now, thanks to the *alloc* instruction in A, 96 registers are allocated for every basic block in G, the program is correct and f can be executed. The allocation value of the *alloc* instruction was chosen as equal to the maximum lrr value of all the basic blocks in G. This is the *normal* usage of the *alloc* instruction described in the Intel Itanium Architecture manuals[6–8]. Note that the *alloc* instruction must be executed before any other instruction that uses a stacked register is executed.

Unfortunately, depending on the control-flow path to be executed, we may allocate more registers than actually required and therefore trigger unnecessary memory traffic. For instance, if the control-flow path $[A, B, D, G]$ is executed at run-time, 96 registers are allocated but only 10 are used. It would be more efficient if,

Criterion 2: For every control-flow path of G, we do not allocate more registers than actually required.

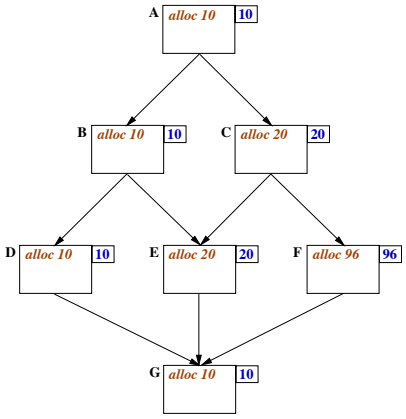

Fig. 4. An *alloc* instruction is inserted in every basic block of G to allocate the exact number of registers required by any basic block.

Using this criterion, we could have the other extreme for the insertion of the allocation instructions shown in Figure 4.

Now, we have satisfied *Criteria 1* and *2* but we, obviously, used unnecessary *alloc* instructions that create a non-negligible increase on the code size and will slow down the program. For instance, the *alloc* instruction in D is redundant and could be removed because basic block B, the only parent of D, already allocated enough registers for D to execute. Thus, we need another criterion to generate an efficient alloc insertion,

> **Criterion 3:** In any control-flow path in G, only "necessary" *alloc* instructions are inserted.

To apply criterion 3, we try to use a simple algorithm that eliminates the *alloc* instruction from a basic block v_i if all the paths that lead to v_i have allocated enough registers to satisfy the *lrr* of v_i. Unfortunately, as shown in Figure 5, this algorithm fails to satisfy *Criterion 3*: for the control-flow path $[A, C, E, G]$, the *alloc* instruction in basic block E is not necessary. However our algorithm failed to eliminate that instruction because the control-flow path $[A, B, E, G]$ does not allocate enough registers before reaching E.

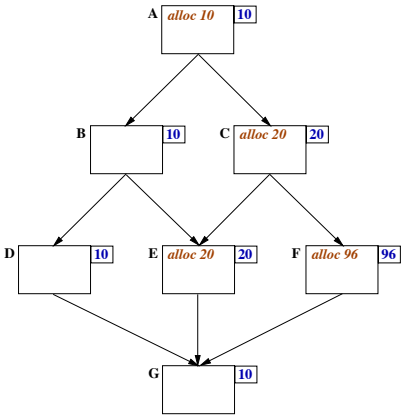

Fig. 5. The *alloc* instruction are all necessary except the one in E for the control path $[A, B, E, G]$.

We want to move the *alloc* instruction that is in E in Figure 5 to another place, so that it is not executed in the path $[A, C, E, G]$. On the other hand, we do not want to move the *alloc* from E to B, because in that case we would allocate too many registers (20 instead of 10) for the path $[A, B, D, G]$ and violate *criterion 2*. Thus, we would like the *alloc* instruction to appear between B and E. We insert an artificial basic block in the CFG when we have no place for the *alloc* instruction that would satisfy all three criteria. Because the allocation instruction insertion phase occurs late in the code generation phase, the insertion of an artificial basic block in the CFG can be costly in term of updating hyperblocks, scheduling, and live ranges. Therefore we need another criterion for our definition of efficient allocation insertion in order to ensure that this method is only used as a last resort,

> **Criterion 4:** The number of artificial basic blocks inserted in G is minimum.

3.2 Problem Formulation

In practice we are concerned with the number of *alloc* instructions executed at runtime, therefore when applying *Criterion 3* we want to take into consideration the frequency of execution of each control path. Thus we now assume that we are also given a function $w(C)$ that specifies the frequency of execution of the control path C when the function f is executed. To implement the third condition we define $N(G, w)$, the number of alloc instructions executed for the control flow graph G under the frequency of execution w as:

$$N(G, w) = \sum_{C \in P(G)} \sum_{v_i \in C} w(C).has_{alloc}(v_i)$$

where $P(G)$ is the set of all control-flow paths of G, $v_i \in C$ indicates that the basic block v_i is in part the control path C, and the function $has_{alloc}(v_i)$ returns 1 if the basic block v_i contains an *alloc* instruction and 0 otherwise. Because the *alloc* instruction can have a long latency (RSE spills/restores) and because the *alloc* instruction introduces new false dependencies with the instructions using the registers being allocated, the less *alloc* instructions executed at run-time, the better (*Criterion 3*). We can now present our problem statement.

Multiple Alloc Problem Statement: Given an acyclic control-flow graph $G = (V, E)$ for a procedure f, a register assignment for the variables of f, and a frequency of execution $w(C)$ for each control-flow path of f, find an allocation instruction insertion scheme A of G such that all the following conditions are satisfied:

(i) **Correctness Criterion:** for every control-flow path C of G, enough registers are allocated to allow the correct execution of each basic block in C.

(ii) **Fitness Criterion:** for every control-flow path C of G, the number of registers allocated does not exceed the maximum local register requirement of any basic block in C.

(iii) **Efficiency Criterion:** the average number of *alloc* instructions executed at run-time, $N(G, w)$, is minimized.

(iv) **Sparseness Criterion:** the number of artificial basic blocks inserted in the CFG is minimized

---

The criteria of the problem statement are sorted in decreasing priority order. For instance, the efficiency criterion must be satisfied before trying to satisfy the sparseness criterion.

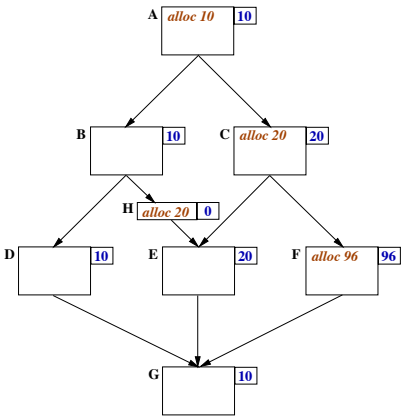

Fig. 6. A solution to our example

In Figure 6 we provide a solution that satisfies all the requirements of our problem statement. A control-flow path may include more than one *alloc* instruction, because a given basic block may belong to multiple control paths. The artificial basic block H has been inserted between B and E to allocate 20 registers for the execution of E without the instruction interfering with the control-flow path $[A, C, E, G]$. *Criteria 1-3* are satisfied while the number of artificial basic blocks inserted is minimized. In any control path, when the flow of execution reaches a basic block, either enough registers have already been allocated or the basic block contains the appropriate *alloc* instruction. Also, after any *alloc* instruction that allocates r registers, at least r registers are actually used later in the CFG. Finally, $N(G, w)$ is minimized and equal to $4 + 2 + 1 + 1 = 8$ when considering the control-flow paths in the following order: $[A, B, D, G]$, $[A, B, H, E, G]$, $[A, C, E, G]$ and $[A, C, F, G]$.

4 Solution Method

In this section we introduce an heuristic algorithm that generates a multiple *alloc* instruction placement. Given a CFG annotated with the lrr for each basic block, this algorithms finds a set of *alloc* instructions that satisfies criteria 1-3. Although this algorithm does not minimize the number of artificial basic blocks inserted, our observation indicates that few such blocks are actually inserted in the code. In its current formulation, the algorithm assumes that all control-flow paths have the same frequency of execution.

4.1 The Algorithm

Before inserting the *alloc* instructions, the following intermediate values need to be computed.

$lrr(A)$**:** Local Register Requirement of basic block A. As defined earlier, it is the maximum number of live registers at any point in basic block A. To be executed, A requires that $lrr(A)$ registers be allocated on the stack.

$orr(A)$**:** Outgoing Register Requirement of basic block A. It is the minimum number of stacked registers required by any control-flow path in the CFG that originates in A, A included, and ends at the exit node. The $orr(A)$ can be defined recursively by: $orr(A) = max(lrr(A), min(orr(S_1), \ldots, orr(S_n)))$ where the $S_1, \ldots, S_n$ are the direct successors of A in the CFG.

Given a register assignment, the $lrr(A)$ and the $orr(A)$ are intrinsic properties of A. The following two values are determined by the placement of *alloc* instructions:

$alloc(A)$**:** Number of registers allocated by the *alloc* instruction in A. If there is no *alloc* instruction in A, then $alloc(A) = 0$.

$maa(A)$**:** Minimum Actually Allocated. The value represents the minimum number of registers actually allocated in any control-flow path that originates in the start node of the CFG and ends in A, A included. The $maa(A)$ can be defined recursively by: $maa(A) = max(alloc(A), min(maa(P_1), \ldots, maa(P_n)))$ where $P_1, \ldots, P_n$ are the direct predecessors of A in the CFG.

For performance, multiple *alloc* instructions should not be placed inside loop nests. Therefore, for the multi-*alloc* algorithm, each loop nest is represented as an aggregate node in the CFG, *i.e.*,a single virtual basic block with a single set of values (lrr,orr,...). The lrr of a loop nest is the maximum register requirement of all the basic

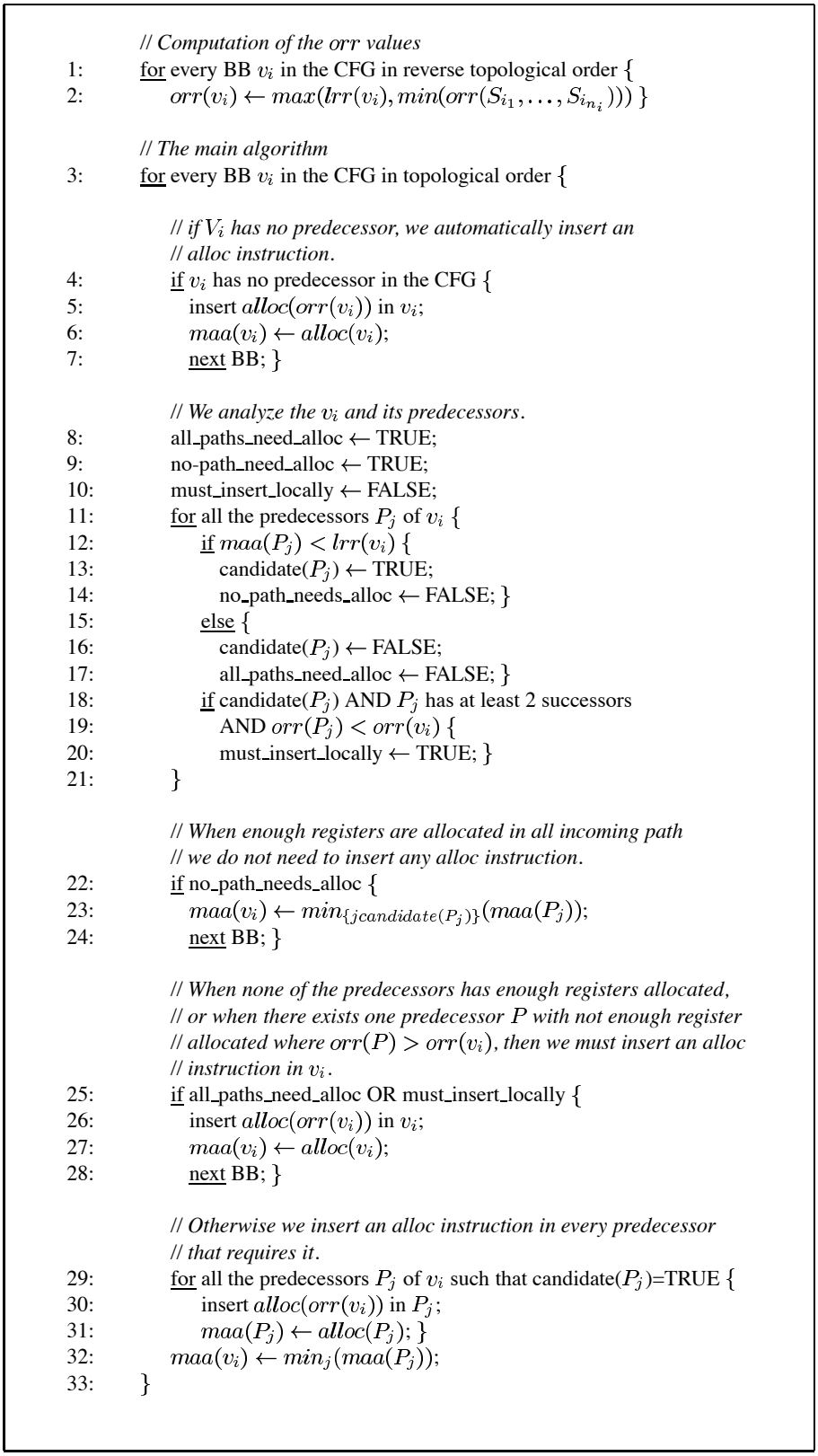

Fig. 7. The multi-*alloc* placement algorithm

blocks in the loop nest. This is a conservative approach to loop nests, but effective in practice.

Before applying our algorithm, we inserted empty basic blocks on the entrance edges of loops to make sure the algorithm is able to insert *alloc* instructions in the predecessors of loop entry basic blocks if necessary. The inserted basic blocks have only one successor and therefore the insertion of an *alloc* instruction is compatible with the fitness criterion.

Our multi-*alloc* placement algorithm is shown in Figure 4.1. First (lines 1-2), a bottom-up topological traversal is performed to compute the orr values using the lrr values.

Then each basic block is considered in topological order (line 3). If the basic block has no predecessor in the CFG, we insert an *alloc* instruction in the block (lines 4-6). If the basic block requires zero stacked registers, insert $alloc(orr(v_i))$ is converted into a no-operation.

Given a node v_i, we check all the immediate predecessors of v_i to identify which ones are candidates for the placement of an *alloc* instruction (lines 11-16). A predecessor P_j of v_i is a candidate for an *alloc* placement if its $maa(P_j)$ is smaller than $lrr(v_i)$.

If all the incoming paths of v_i need an *alloc* instruction, then the *alloc* instruction is placed in v_i itself (lines 19 and 25). The number of registers allocated is equal the maximum number of registers that will be required in any path leaving $v_i, orr(v_i)$. By allocating $orr(v_i)$ instead of $lrr(v_i)$, we prevent the need for the insertion of another *alloc* instruction in at least one path leaving v_i.

If there exists at least one incoming path that does not need the *alloc* instruction, the algorithm inserts one *alloc* instruction in each of the incoming paths that need it (lines 27-29). Finally we update the $maa(v_i)$ value (line 30).

4.2 Application to our Motivating Example

The application of the algorithm to our motivating example is shown on Figure 8. In this figure each basic block is annotated with its lrr, orr, and maa values. Figure 8(a) shows the CFG after the computation of the orr values(lines 1-2). Then the CFG is traversed in topological order (line 3). The first basic block, A, has no predecessor therefore we insert an *alloc* instruction (line 4-6) with $alloc(A) = orr(A)$ (Figure 8(b)). Next basic block B is visited. Because $lrr(B)$ is equal to $maa(A)$ (an immediate predecessor of B) no *alloc* instruction is needed in B (lines 11,14-15). Assume that the algorithm visits C next. A is the only predecessors of C, and $maa(A)$ is smaller than the $lrr(C)$. Therefore an *alloc* instruction must be inserted in C. Because A, the only predecessor of C, has a lower orr value, the insertion must be in C (lines 17-19, Figure 8(c)). When the algorithm visits D, its only predecessor B has enough registers allocated ($lrr(D) \leq maa(B)$) (lines 11,15-16), thus D does not need an *alloc* . As for B we do not insert an *alloc* instruction in D. E has two predecessors and only one incoming path, from B, requires the insertion of an *alloc* instruction. Because $orr(B) < orr(E)$, the insertion must be local (lines 17-19, Figure 8(d)). Then the algorithm continues and an *alloc* instruction is inserted in F but not in G (Figure 8(e)).

4.3 Algorithm Analysis

Time Complexity

Theorem 1. *The algorithm is linear in the number of basic blocks in the CFG.*

Proof. The algorithm traverses the CFG in topological order and only visits the predecessors of every basic block. Visiting a predecessor is equivalent to following an edge backwards. In a CFG each node can have at most 2 immediate successors. Thus the number of edges in a CFG is proportional to the number of nodes. Therefore the entire loop can be executed in linear time in the number of nodes.

For the same reason, the insertion of artificial basic blocks does not change anything to the time complexity. □

Criteria Satisfaction First we prove that the two first criteria of our problem statement are satisfied.

Theorem 2. *The algorithm proposed returns an allocation instruction insertion scheme that satisfies the correctness criterion.*

Proof. The algorithm traverses the graph in top-down topological order. For each basic block, the algorithm tests if enough registers have been allocated for every incoming path to the basic block (line 11). If an *alloc* instruction is required to satisfy the local register requirement of the basic block, the algorithm inserts one either directly in the basic block (lines 4-6 or 24-26), either earlier in the faulty paths (lines 27-30). Therefore the correctness criterion is satisfied.□

The fitness criterion is not satisfied as our example shows for the basic block E in Figure 8(e). If the control-flow paths comes from C, then we repeat the *alloc* instruction.

Because the algorithm does not take into account the frequency of execution of any given control-flow paths of

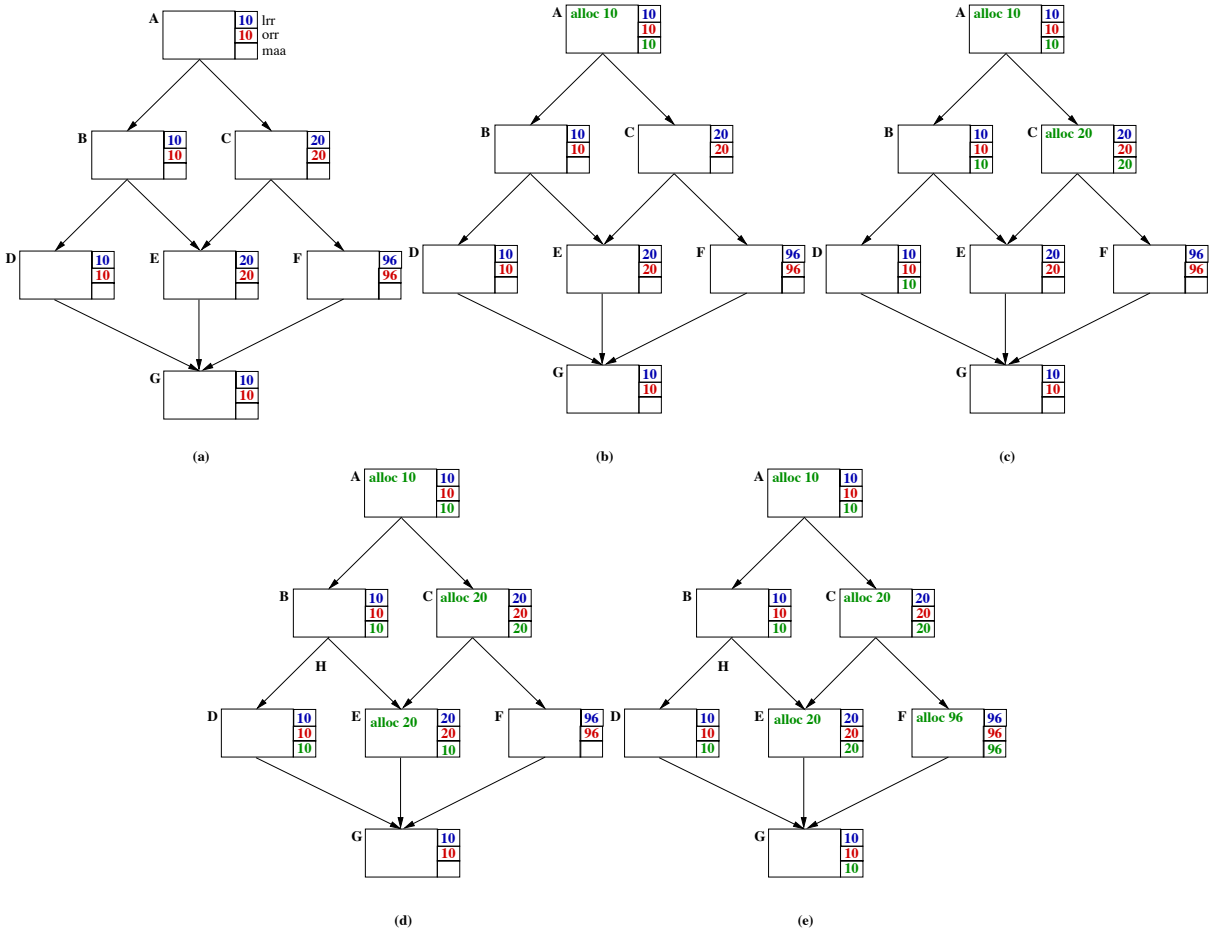

Fig. 8. Application of the algorithm on our motivating example.

G, the algorithm cannot return an allocation instruction insertion scheme that satisfies efficiency criterion. However, if each control-flow path has the same frequency of execution, then we believe that the algorithm satisfies the criterion in most of the cases.

Since we do not insert new basic blocks in the CFG at all, the number of inserted basic blocks is obviously optimal.

5 Experiments and Results

5.1 Experimental Framework

We implemented the multi-*alloc* algorithm with the two optimizations in the industry-strong Open64 compiler ([1,2]. The *alloc* instructions are inserted right after the register allocation phase but before the last instruction scheduling phase of the compiler. Our experiments were performed in an HP workstation i2000 equipped with a single 733MHz Itanium processor and 1GB of memory and running Debian Linux 2.4.7.

Currently we have tested the implementation on 7 SPEC CPU2000 benchmarks programs.[3] We measured the number of *alloc* instructions inserted and the number of registers saved due to our algorithm. We compare our results to the standard algorithm for the *alloc* instruction, *i.e.*,an algorithm that inserts a single *alloc* in each procedure entrance. On average, we allocate 1.38 less registers per procedure with a maximum of 26.50 registers saved. We use 1.91 *alloc* instructions on average with a maximum of 32 instructions in a procedure. This average is weighted by the frequency of execution of each basic block.

[3] We expect to include more benchmarks to this list by the camera ready submission deadline.

Benchmark	Average number of *alloc* inserted	Largest number of *alloc* inserted	Absolute number of registers saved	Relative number of registers saved	Best number of registers saved	Execution time
164.gzip	1.92	11	1.31	14.50%	5.16	+27.87%
175.vpr	1.74	32	1.46	13.06%	15.78	+5.36%
181.mcf	1.26	4	1.50	14.93%	6.44	+61.98%
186.crafty	2.77	25	1.66	20.10%	19.95	+30.39%
254.gap	2.50	23	1.20	16.98%	3.61	+22.47%
256.bzip2	1.63	12	1.20	14.81%	3.01	+??%
300.twolf	1.53	17	1.33	15.54%	26.50	+??%
average	1.91	*	1.38	15.70%	*	+??%

Fig. 9. Number of registers saved and *alloc* instructions inserted for each of the seven benchmarks tested.

5.2 Implementation Considerations

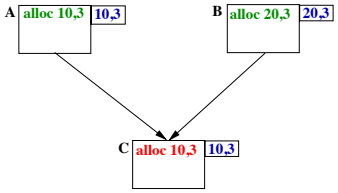

Fig. 10. Necessary extra *alloc* instruction for function calls

For simplicity, the algorithm presented in this paper assumes that the *alloc* instruction has a single parameter, *i.e.,* the size of the current register stack frame. However, in the *alloc* instruction in the Itanium architecture specifies the number of input, output, local, and rotating registers. Thus an implementation of the algorithm has to include different strategies for each type of register.

The rotating registers overlap with the local and output registers. In our current implementation, the *alloc* instructions requests rotating registers only when the number of rotating registers required is less than the sum of local and output registers. If this is not the case, then obviously there can be no downstream loop that uses rotating registers. A downstream loop that requires rotating registers would have been taken into account in the *orr* values.

The input registers are easily handled because they are part of the local section of the register stack frame. Input registers are used to specify how many registers in the new stack frame overlap with the previous stack frame.

The local registers and the output registers were the only types of registers that require modifications to the simplified algorithm. Each basic block needs the full set of values (lrr, orr, maa and $alloc$) for each of the two types of registers. Thus, an *alloc* instruction is inserted in a basic block if either local or output registers are required (OR statement).

In some cases the introduction of a second parameter forces us to insert more *alloc* instructions to preserve the correctness of the program. This situation happens when two different control flows reach a function call and the number of local registers allocated in each incoming path is distinct. Consider, for instance, the example shown in Figure 10. Each *alloc* instruction is annotated with two numbers: the number of local registers to the left, and the number of output registers to the right. Because of automatic register renaming, r32 is always the first register in a stack frame. If we consider only the number of registers required in each basic block, the *alloc* instruction in basic block C is not necessary because there are enough registers allocated in either incoming path. However, if block C has a function call that expects three output registers, there is a problem: the boundary between local and output register depends on the incoming path: if we reach block C from block A, then $r42$ is the first output register. Whereas, if the flow comes from B, the first output register is $r52$. Therefore, at the function call site, there is no way to tell at compile time which register is the first output register. We must insert an *alloc* instruction before the function call to ensure that the output registers start at $r42$ regardless of the incoming path as shown in our example.

5.3 Results

Table 9 shows our results for the seven benchmarks tested. These numbers are weighted by the frequency of execution of each basic block in the routine. Thus the basic blocks and *alloc* instructions in a control-flow path that is executed 1 out of 10 times that the routine is executed is weighted by 0.1. Then we take the average for all the routines in the benchmark.

The number of registers saved can be significant with a maximum of 26.50 registers for one routine of 300.twolf. The number of registers not allocated thanks to our optimization is low: 1.38 on average. Nonetheless, the algorithm reduces the register stack frame size of a routine by 15.50%, on average. These results are explained by the relatively low register pressure in the SPEC2000 benchmarks.

Although the algorithm does not try to limit the number of *alloc* instructions inserted in a given routine, the average number of instructions inserted is 1.91. By adding one more *alloc* instruction per routine, we can manage to reduce the number of registers allocated by 15.70%. However, in some rare cases the number of *alloc* instructions inserted is high. For instance in one routine of 175.vpr the algorithm inserted 32 *alloc* instructions. We are investigating such cases to identify opportunities for improvement.

Despite the savings in register allocated, the execution time of the programs has been increased by up to 56% for *crafty*. The main reason is the cost of the *alloc* instruction itself that was not taken into account by the algorithm. This instruction is expensive and introduces false dependences that can break a good instruction schedule. Moreover the insertion of basic blocks to host *alloc* instructions at the entrance of loops results in the insertion of branch instructions as well. Finally, most of the time, the difference between the number of registers allocated between two *alloc* instructions is small and inserting a second *alloc* instruction does not pay off.

6 Future Work

Future generations of processors of the Itanium family are expected to have a much more efficient Register Stack Engine. We anticipate that the implementation of eager spill and eager fill modes in the RSE will lead to a more effective application of the idea of using multiple *alloc* instructions introduced in this paper. Moreover we plan to study the following modifications to the original algorithm:

- The algorithm could use an *alloc* instruction immediately before a loop entry to reduce the number of registers allocated to the number of variables live at that point in the program, and another *alloc* instruction at the loop exit to restore the number of registers required by the paths that leave the loop.
- A similar solution could be used around function calls. In the Itanium architecture, there are 96 registers available in the register stack. As long as all the cumulative number of registers requested by active functions is less then 96, there will be no spills and fills. Using this observation, we could delay the time when the 96 register threshold value is reached by shrinking the current register stack frame as much as possible right before every function call.
- When feedback profiling information is available, the multiple *alloc* placement algorithm can favor placing the least number of *alloc* instructions in the control paths that have the highest frequency of execution. The placement of *alloc* instructions in other paths would be secondary to this constraints.
- the insertion of *alloc* instructions could be triggered by a profitability analysis, and be restricted to the places where the gain is significant enough. The registers could be allocated in chunks, or *quanta*, of 5 or 10 up to the maximum needed by the function. It would reduce the number of *alloc* instructions in the program (efficiency criterion) with a limited cost for the fitness criterion. This idea follows from the implementation of efficient dynamic memory allocation algorithms.

7 Acknowledgments

We would like to acknowledge Gerolf Hoflehner and Jim Pierce for their contributions and for insightful comments about our approach to the problem. This research is supported by the National Science Foundation (NSF), by the National Security Agency (NSA), by the Defense Advanced Research Projects Agency (DARPA), and by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

8 Conclusion

In this paper, we tried to solve the problem of inserting *alloc* instructions in Itanium code in order to achieve a finer-grain allocation scheme and reduces the number of blocking spills and restores with the register stack engine. We defined four subgoals: correctness, fitness, efficiency and stability and proved that the problem was NP-complete.

Then we propose a heuristic that solves the first two criteria: correctness and fitness. The algorithm is linear and achieves ...

However the algorithm did not consider the frequency of execution of the control-flow paths in the CFG and the resulting code could be further improved. Also, the eager allocation modes were not available in the Itanium processor used for the the experiments, although it would a efficient source of improvement.

The next step is now to consider the frequency of execution of control-flow paths and try different levels of optimizations by releasing the fitness constraint for instance.

References

1. Open research compiler for itanium processors. http://ipf-orc.sourceforge.net/, January 2002.
2. Open64 compiler and tools. http://open64.sourceforge.net/, January 2002.
3. D. Callahan and B. Koblenz. Register allocation via hierarchical graph colloring. In *SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 192–203, Toronto,ON, June 1991.
4. G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN 82 Symposium on Compiler Construction*, pages 98–105, June 1982.
5. F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Language and Systems*, 12(4):501–536, October 1990.
6. Intel Corporation. *Intel Itanium Architecture Software Manual vol1-4*, December 2001. revision 2.0.
7. Intel Corporation. *Intel Itanium Processor Reference Manual for Software Development*, December 2001. revision 2.0.
8. Intel Corporation. *Intel Itanium Processor Reference Manual for Software Optimization*, November 2001. http://developer.intel.com/design/itanium/.
9. R. Govindarajan, H. Yang, J. N. Amaral, C. Zhang, and G. R. Gao. Minimum register instruction sequence problem: Revisiting optimal code generation for dags. In *15th International Parallel and Distributed Processing Symposium*, San Francisco, CA, April 2001.
10. R. Govindarajan, H. Yang, J. N. Amaral, C. Zhang, and G. R. Gao. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Transactions on Computers*, 2002.