Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 04-044

Dynamic Register Allocation for ADORE Runtime Optimization System

Aditya Saxena and Wei-chung Hsu

December 01, 2004

Dynamic Register Allocation for ADORE Runtime Optimization System

Aditya Saxena, Wei-Chung Hsu
(asaxena, hsu@cs.umn.edu)

Abstract

Dynamic optimization can apply several difficult to apply optimizations at run time. These optimizations are difficult to apply at compile time because of lack of accurate run time information. A dynamic optimizer can make use of run time characteristics to identify optimization opportunities. However, to deploy the optimizations, extra registers are required. In this report we explore methodology and issues involved in techniques that can be used to allocate registers dynamically and present experimental results. The techniques that have been explored include dead register detection, register spilling and using IA64 specific alloc instruction under the ADORE framework.

1. Introduction

Dynamic optimization offers several opportunities for applying optimizations that are typically difficult to apply at compile time. For example compilers generally schedule the code based upon the assumption that data is present in the cache. This assumption may not be true leading to inferior run time behavior. At higher level of optimizations the compiler aggressively applies optimizations such as data pre-fetching, which at times can result in a slowdown. This may happen because of cache pollution and extra cycles introduced in loops. In other cases it may be impossible to apply any optimization. For instance, shared libraries are loaded into process address space at run time. This

precludes several important optimizations such as procedure inlining, which provides other important static optimization opportunities such as dead code elimination, instruction scheduling, copy propagation, common sub-expression elimination, etc.

1.1 Need for dynamic register allocation

A dynamic optimizer can optimize a binary based upon runtime characteristics. However to deploy the optimizations, the optimizer needs extra registers. The optimizer may be targeting to reduce data latency by issuing a prefetch instruction for the data earlier in the code sequence. However the data address needs to be calculated in a register. The dynamic optimizer may also speculatively execute instructions, for which even larger number of free registers is needed. Registers already present in the code sequence holding live values cannot be used. Hence registers need to be allocated dynamically to apply the optimizations.

In this report we discuss the methodology and issues involved in dynamic register allocation using various techniques such as finding dead registers, register spilling and using some IA-64 specific features for the ADORE [1] runtime optimization system. IA-64 related methods involve the use of *alloc* instruction present in the ISA and manipulating Register Stack Engine (RSE) to bump up the process register stack. Average lifetime of dead registers has been evaluated for several SPEC2000 benchmarks. Dead register evaluation has been done on traces [2] generated from binaries by running them under the ADORE framework. We also evaluate the percentage of registers that can be spilled in a given trace.

The rest of the report is organized as follows. In the next subsections we give a brief description of the ADORE dynamic optimization system, trace generation and IA-64 architecture. In the next section we explore the register allocation techniques. This is

followed by conclusion and references.

1.2 ADORE Framework

ADORE [1] (Adaptive Object code RE-optimization) is a dynamic optimization system. It is implemented on IA-64 architecture. It is a shared library that is loaded with the application at startup and executes as a separate thread. The system uses Itanium Hardware Performance Monitoring (HPM) [8] unit to collect information about the runtime characteristics of the binary. In particular, it uses Branch Trace Buffer (BTB), which is configured to record four most recent branch source/targets. This information is used to build traces [2] that represent hot code in the current execution time interval of the program. ADORE defines traces as single entry multiple exit sequence of code that could range from a few to tens of basic blocks. The system optimizes these traces, places the optimized traces in the code cache, and patches the entrances to the traces in the original binary by a branch to the optimized traces in the code cache. Earlier ADORE used to reserve registers in the compiler for deploying optimizations. This technique however was not transparent. Since it is unlikely all compilers are willing to support ADORE type of dynamic optimizer. This also leaves less registers for use by compiler. The approach of register reservation is also less flexible in that some long traces may need a lot of registers to perform optimization while some simple traces need only a few registers. Allocating a fixed number of registers will be insufficient for some cases and be wasteful for other cases. Later ADORE started to exploit the *alloc* instruction in IA-64 to bump up the size of stacked register window allocated by the compiler. This technique though applicable to most cases, lacks generality, as there are some corner cases in which it is either not applicable or may lead to inferior performance.

1.3 Brief Description of the IA-64 Architecture

IA-64 [5] is a RISC based, register-register instruction set. It has several features such as predicated execution, memory reference speculation, and massive resources in terms of integer and floating point register file. Both these register files support register rotation, which makes software-pipelined loops easier to implement. These features make it suitable to exploit compiler-based exploitation of ILP. The HPM unit provides hundreds of counters which provide runtime information about executable such as CPU cycles, data cache miss addresses, load latencies etc, which makes it a suitable platform for implementation of dynamic optimization systems. The techniques discussed in the next section are discussed with respect to this architecture. The details of specific features of the architecture are provided wherever an assumption is made about the underlying platform.

2. Dynamic Register Allocation Policies

In this chapter we discuss various dynamic register allocation policies, explaining the methodology and issues with experimental results.

2.1 Dead Registers

A register is said to be dead in a trace between its last usage and its redefinition or a definition and a redefinition without any intervening usage. The usage location is defined as one having any instruction in which the register is used as a source operand. Redefinition is the location where an instruction uses the register as a destination operand, thereby destroying the old value. Hence a register cannot be considered dead if there is no redefinition of the register at any point in the trace. The register is effectively free for usage by the dynamic optimizer in the detected range. It can be assumed all registers are defined at trace entry, that is, they hold live values.

There are several issues in the detection and usage of dead registers. Some of these issues arise because of the IA-64 instruction set architecture while others are more general language and system dependent issues. We first discuss the instruction set related issues and then language and system issues.

2.1.1 Instructions with side effects

IA-64 instruction set has some instructions that have side effects, i.e. they modify their source operands. A register can be used and defined by the same instruction. These instructions fall into the category of integer and floating point load/store and lfetch [7] instructions with base update form. The lfetch instruction is used to prefetch a cache line to reduce load latency. The base may be updated through a register or an immediate. The load instructions are ld, ldf(floating point load) and ldfp(floating point load pair). A pseudo load format can be written as

(qp) ld r1 = [r2], addend

Here the addend parameter (which is optional) could be a register or a signed immediate. Register *r2* is atomically updated with addend. Hence we first include a usage of *r2* and then a definition for this instruction location.

The store instructions are st and stf(floating point store). A pseudo store format can be written as

(qp) st [r1] = r2, addend

Register *r1* is atomically updated with the value of addend. Hence we include a usage of *r1* and then a redefinition for this instruction. The same holds for line prefetch instruction.

2.1.2 Branch instruction [7]

The traces we consider in this study are single entry, multiple exit sequence of instructions. This implies that there could be a branch (direct, indirect, function call or return) within the range of register usage and redefinition. To examine their effect we consider four kinds of branches.

2.1.2.1 A normal branch

This is a predicated conditional branch which is not taken frequently. If it's unconditional, i.e. un-predicated branch and transfers control out of the trace then the branch target should be part of the trace. If it is in trace, then it should be part of a loop or a forward jump.

2.1.2.1.1 Off-trace Branch

If the branch target is off trace we must assume a usage of all registers at this point. To see why this is necessary consider the case shown below.

R (used) A

:

Offtrace branch B

:

R (defined) C

We cannot consider register R dead from A to C because of the intervening branch. The branch target instructions could potentially read R resulting in incorrect execution if we had modified R. Hence the conservative and correct approach would be to consider R dead from B to C.

2.1.2.1.2 In-trace branch

We consider the following cases for backward branches (loops) and forward branches. In trace branch here does not mean a jump into a trace from outside.

2.1.2.1.2.1 Forward jumps

These branches can exist inside a loop which take control out of the loop or part of an IF statement. Since we cannot know whether the branch condition is true or not we assume that branch is never taken. This constricts the range of a plausible dead register but preserves correctness.

2.1.2.1.2.2 Backward branches (loops)

Let's assume a trace divided into blocks of instructions as shown in figure 1.

Region A includes the entry portion of the trace and may be empty. B is the loop target; C is the loop body, in which we assume there are no backward branches for sake of simplicity. D is the conditional branch and E consists of instructions after the loop.

The following cases exist (we assume all registers have a definition before the beginning of the trace).

a) A register is defined/used in region A and is used in B or some instruction in C but not redefined. In this case the register is not dead anywhere within the loop. However, if it has a redefinition in E, then its can be assumed to be dead from the beginning of E till the definition. Assuming it dead from its usage in C and modifying it would cause incorrect execution of loop body. If it is not redefined in E, then we cannot assume it dead anywhere in the trace.

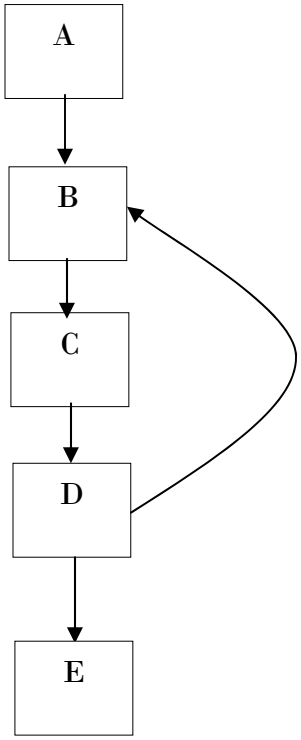


Figure 1: A, C, and E represent blocks of instructions without branches. D is a branch and B is the target.

b) A register is defined/used in region A and is redefined in B or C before the usage. In this case the register is dead from the last definition/usage in A till the redefinition in B or C. The register is dead from last usage till the end of the loop but only in case the branch is taken. In static analysis we cannot be sure of this condition. However the last usage point can be assumed to be in region B/C as opposed to that in case a). Hence this case does not need special treatment from the straight-line code analysis without any branch.

c) A register is defined/used in region A and is used in B or C before redefinition in C. In this case the register is dead between the last usage in loop and the redefinition. The redefined value flows to the usage in the next iteration or outside the loop. This case too does not need special handling as in b).

d) The register is not used at all in B or C. If the redefinition appears in E then the register is dead throughout B and C. However this too doesn't need special handling.

This discussion thus suggests that for loops only case *a)* need to be considered. In region C we assume there can be no backward branches. We do not consider multiply nested loops since they are rare in traces.

2.1.2.2 Procedure call instruction

This branch instruction is used to affect a procedure call in IA-64. This branch effectively saves the *caller*s stacked register frame. The called function is provided with a register frame consisting of only output registers of the calling procedure i.e. the output registers are visible across the procedure call. These output registers could be used as scratch registers by the called procedure. This introduces two problems with respect to correctness of using dead registers and the intended optimization. 1) We may detect an output register as dead and modify it resulting in wrong parameter being passed to the *callee*. 2) The called function may change the value of a dead register during the call in which we may have stored some value for use after the call. This would affect any optimization we intended using this register. This is not a very serious problem if the optimization is architecturally safe. In worst case the optimization may just fail.

In absence of any knowledge about the stacked register configuration of the calling procedure we need to be conservative and ignore the trace having a procedure call. This is the approach we have used in this study. However if we know the register stack

configuration, which can be determined by examining the *alloc* instruction parameters, then we would be able to make intelligent decisions and consider only the input and local registers for "deadness". We also don't consider static registers because of issues with exceptions discussed later. Table 1 shown below gives the number of traces having a call instruction and the total number of traces created during each phase of execution of the binary. The binaries have been compiled by ORCC with O2 option.

Table 1. Number of traces with calls/total number of traces detected

Benchmark	***Calls/Trace***
Ammp	0/3, 0/9
Bzip	4/42,8/21
Equake	3/26
Gap	2/30,30/65,13/22,12/23,12/23
Gcc	27/120
Mesa	8/48
Twolf	5/37
Vpr	8/28
Mesa	11/49
Parser	41/120

As the data suggests there is a significant amount of function calls in bzip, gap, gcc, parser, and vpr traces.

2.1.2.3 Function returns

This branch affects a return to the calling function. This is an off-trace branch and we can effectively ignore it since if executed the register stack would go out of scope.

However for the sake of correctness we must assume a use of registers r8-r11 at the location of this branch since these registers are used to return values to calling function. Failing this we may end up modifying these registers destroying the return values.

2.1.2.4 Explicit Loops

IA-64 provides special branch modifiers for counted loops and instructions for software pipelining and modulo scheduled loops [5]. The stacked registers are renamed by hardware in software pipeline and modulo schedule enabled loops, during each iteration. Hence it is not safe to deal with registers inside these loops. Thus we ignore the traces with branches controlled by wtop, wexit, ctop and cexit modifiers. However loops controlled by cloop modifier (counted loop) can be taken into consideration and treated in a similar way as described for normal branch controlled loops. Table 2 below shows the number of software pipelined/modulo scheduled loops and the total number of traces with loops.

Benchmark	Special Loops/Total Loops
Ammp	0/2,4/7
Bzip	11/15,3/4
Equake	6/11
Gap	10/10, 5/6,0/0,0/0,0/0
Gcc	15/38
Mesa	7/8
Twolf	18/18
Vpr	1/8
Parser	17/31

Table 2: Number of traces with special loops and total number of loops in all the traces.

To summarize, for off trace branches, we assume a usage of all the registers at the branch

point. We ignore the traces having functions calls. We make a usage of r8-r11 at function return. We ignore traces with software pipelined and modulo scheduled loops. For normal and *cloop* controlled loops a special case is needed only when there is no definition inside the loop body. Forward jumps are assumed to be not taken.

2.1.3 Predicated Instructions

IA-64 provides for conditional execution of instructions, which are controlled by a predicate register [5], [6]. The architecture provides 64, 1-bit registers in a predicate register file. If the predicate is true then the instruction executes normally. Otherwise the instruction does not modify the architectural state. A zero predicate is interpreted as false while a one is interpreted as true. Predicate register zero is hardwired to one. The predicates are set mostly by compare instruction *cmp* and *cmp4* [7] that, in the normal form, set two predicate registers to complementary values. Other instructions that set predicates are *tbit, tnat, fcmp, fclass, frcpa, fprcpa, frsqrta* and *fprsqrta* [7]. The last four in the list set only one predicate. For all the instructions except the last four, multiple types of comparison are available which dictate how the values of predicates are set. This implies that both the predicates may be set to the same value.

The presence of predicated instructions not only complicates the detection of dead register, but also reduces the range in which it is dead. As an example consider the straight-line pseudo code fragment shown below.

```
        add r13=r21,r22          //      X
                                 :
(p8)    add r21=r25,r26;;        //      X+m
                                 :
(P9)    add r21=r25,r27          //      X+n
                                 :
        sub r13=r21,r22          //      X+o
```

Here a register is defined at location X and redefined under control of predicates at

locations X+m and X+n. If we don't have any information about the predicates then we are no longer sure which definition is actually taking place, if at all, since both predicates may be false and the original value may flow right up to the usage at X+o. Hence we cannot consider r21 dead anywhere for the sake of correctness. However, even if we don't know the value of predicates, the knowledge that they are complementary can help us reach the conclusion that at least one definition is taking place. We can select the earlier definition as the one taking place and determine r21 to be dead between X and X+m. The actual range may be from X to X+n but we cannot be sure of that. This is certainly better than the earlier case where we are completely clue less. It can be determined whether the predicates controlling definitions of a register are complementary or not by examining the compare instruction, responsible for setting them. The following cases can be made where we cannot accept a definition as taking place.

1) The information about predicates is not available in the trace (i.e. the controlling predicates were set before entry to the trace).

2) The compare instruction setting predicates is of such a type that may set both the predicates to the same value. This occurs for the following types: *unc*, *and*, *or*, *or.andcm*, *orcm*, *andcm*, *and.orcm*.

3) When the instruction setting the predicates sets only a single predicate.

4) When the compare instruction itself is predicated. Here we are not sure whether the predicates will be set or not. This may make us loose prior knowledge about the predicates, except for the case when they are known to be complementary. The property of being complementary will not change whether the instruction is executed or not.

For register usage information we always select the latest usage, either predicated or not. Hence we don't need to make a special case for them.

Algorithm and Implementation: We store the predicate and the location of their defining instruction for each register definition. If the predicates for two definitions are defined at the same location then they are complementary. A list of all the definitions of a particular register between usages is maintained. If we are able to find predicated definitions that are controlled by complementary predicates then the earlier definition is kept else both are deleted. The algorithm can be extended to cases where more than two definitions are controlled by different predicates between usages.

2.1.4 Asynchronous Exceptions

Asynchronous exceptions cause system dependent restrictions on the usage of dead registers. Several cases exist when an asynchronous exception can be generated and delivered to an application. The application may have arranged the delivery by itself (timer), may have been signaled by a co-operating process (kill system call), notified by the operating system of some system event (from keyboard) or it may have been generated by erroneous execution by the application itself (segmentation fault, illegal instruction, etc). We are not concerned about the case where the signal causes process termination, resulting from erroneous behavior or where the default behavior is ignoring the signal or terminating the process. A problem may arise when the application has installed a user defined signal handler. Since the signal delivery is asynchronous, we cannot be sure when it's delivered. The signal handler may use the register we have used as dead and may execute erroneously. The problem however, occurs only if the signal handler is dealing with global variables, which are allocated in a register, definitely a rare condition. The condition must be rare because such a permanent register allocation to a variable must hold throughout the lifetime of the application, because the compiler has no knowledge about whether a function is a signal handler or a normal function and cannot

guarantee whether it will be executed or not. In case of separately compiled modules linked together, this is very difficult. A local variable being allocated to the same register, which was used earlier as dead, is not a problem since the entire register context is created and saved by the operating system and restored automatically on return.

The simplest approach to alleviate the problem would be to refrain from using the static registers i.e. registers 0 to 31 for optimizations. This however decreases the dead registers we may find in a trace. The problem is worsened if the trace belongs to a hot function that is small in size and uses none or only a few stacked registers. The next section shows the percentage of static registers that are dead separately from stacked registers for comparison. The graphs show that the dead registers comprise mostly of static registers in comparison to stacked registers in a large number of traces.

A possible solution against the simple approach of ignoring static registers is to provide information about global variables, to the dynamic optimizer directly from the compiler, which would prevent it from using such registers. Another approach could be to block all the signals. Upon detection of a signal, patch the binary to return to the original code and deliver the signal. Since the asynchronous events are generally rare during the lifetime of an application, the time overhead should not be significant. Because the signal is asynchronous it does not matter when it is delivered, only that it is delivered eventually.

We can also block the signals when we enter the trace and unblock at exit. However this is high overhead since two system calls are required during entry and exit, to prevent against a condition, which is supposedly rare.

2.1.5 Synchronous Exceptions

Languages such as C++ support synchronous exceptions that imply that they can be generated from "throw" expressions. The problem is similar to that for asynchronous

exceptions i.e. only for global variables allocated in register. For example, consider the code snippet shown below.

```
register int i;
try {
        if (error) throw (new error);
} catch (error) {
        recovery_code block;
}
```

In this case if the variable 'i' is allocated to register R and the recovery code is making use of register R, then we cannot make temporary use of R anywhere else.

2.1.6 Experimental Results

In this section we present experimental results regarding dead registers in traces created from applications in SPEC2000 benchmark suite. All the reported binaries have been compiled by ORCC at O2 option. The data indicates average lifetime of a register in a trace. The range over which a register is detected as dead is divided by the number of instructions in the trace (length). This gives the average dead time of a register. Individual contributions from all the registers are then added and divided by the number of registers present in the trace. This figure can be thought of as the probability of finding a dead register in a particular trace that is independent of the trace length and the number of registers present in the trace. Hence it can be argued that a larger percentage is more valuable in terms of optimizations. However, this assertion can be wrong, for example when the optimizer needs to insert a prefetch at a specific location in trace and no register is dead at that location. It does not matter how large the probability is for the entire trace. For example trace 1 for AMMP phase 1 has a probabilty of approximately 9% for finding dead register. The number of instructions in 13 and the number of registers is 8, hence dead registers are present in a total range of ~9 (13*8*9/100) instructions. This range however possibly includes overlaps and is not contiguous. Hence the requirement is

independent of availability affecting the usefulness of dead registers for optimization purposes.

The formula used to calculate the percentage is

$$percentage\ dead\ registers = \sum_i \{ \sum DR_i / I \} / NR \times 100$$

where,

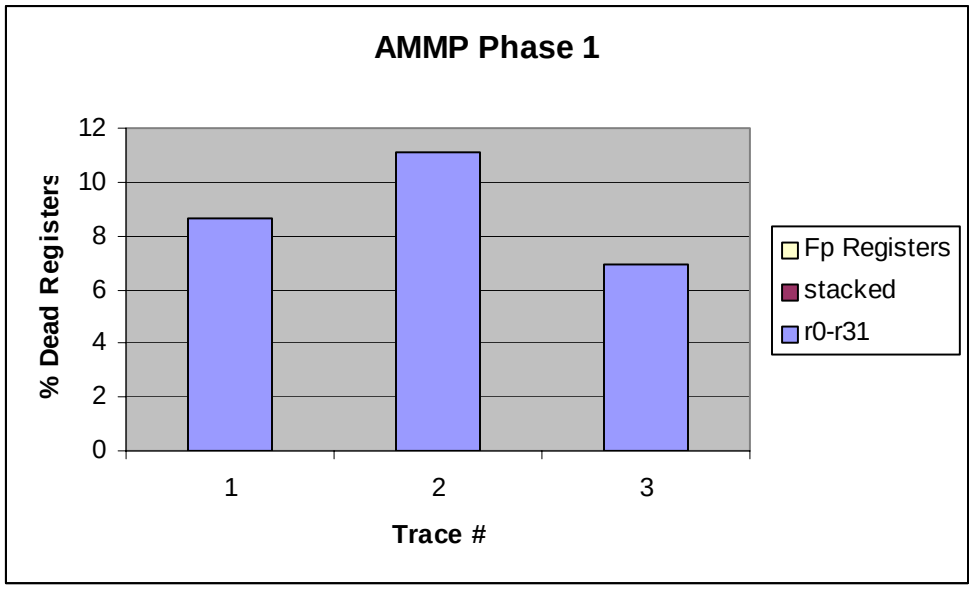
DR_i is the range in which register *i* is dead in trace,

I is the number of instructions in trace,

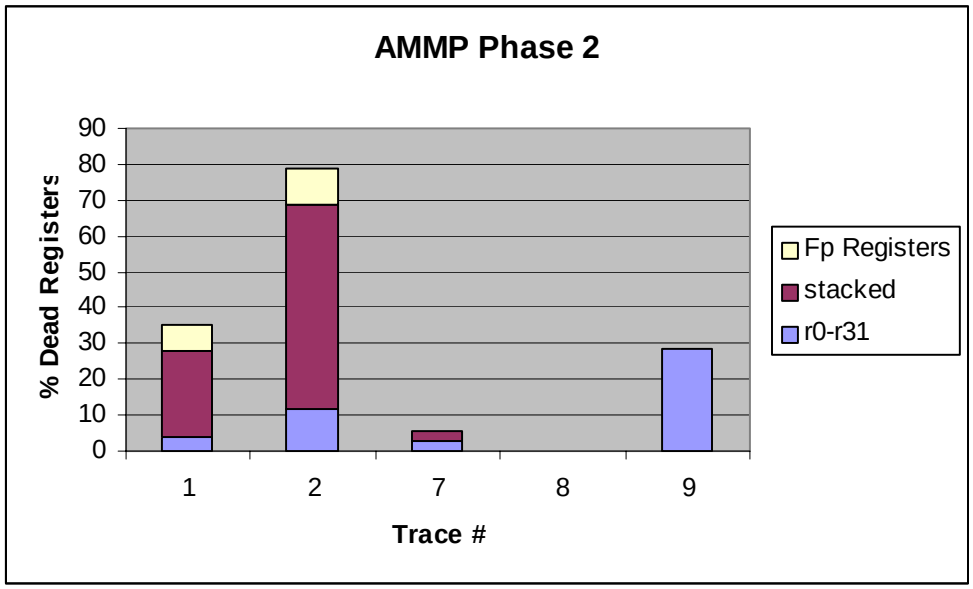
NR is the number of registers referenced in trace, and

i is the summation over all the NR registers.

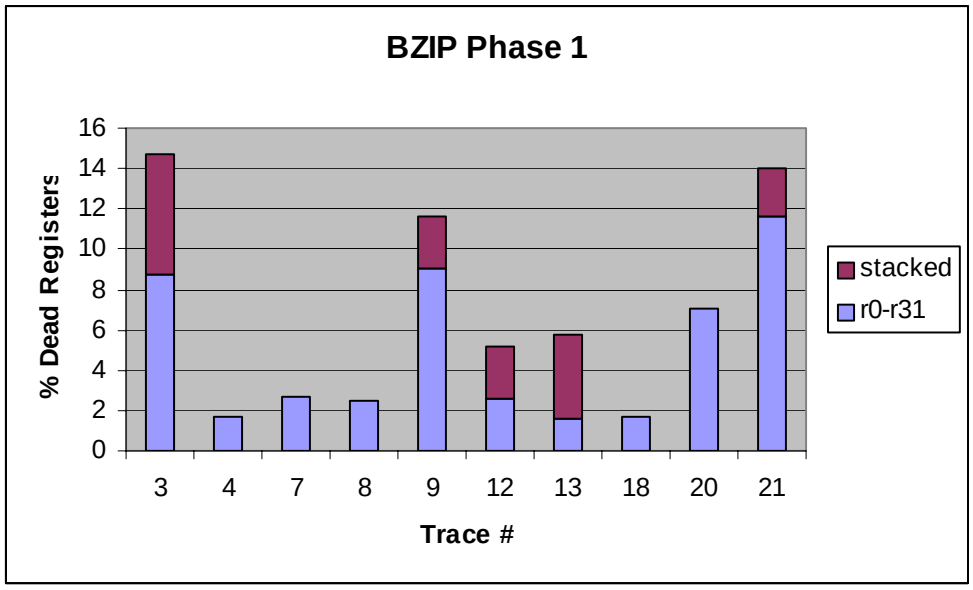
The results have been reported for the top ten traces, where possible, that don't have a function call or a software pipelined/modulo scheduled loop. Only the first two phases are shown for binaries having multiple phases. The floating-point registers, though shown in the same chart have been calculated separately using the same formula. Hence the height of the bars is not indicative of the total number of dead registers in those traces. Instead the difference between the base and top should be taken. The table at the bottom of each chart shows the trace number and the number of registers and instructions present in that trace.



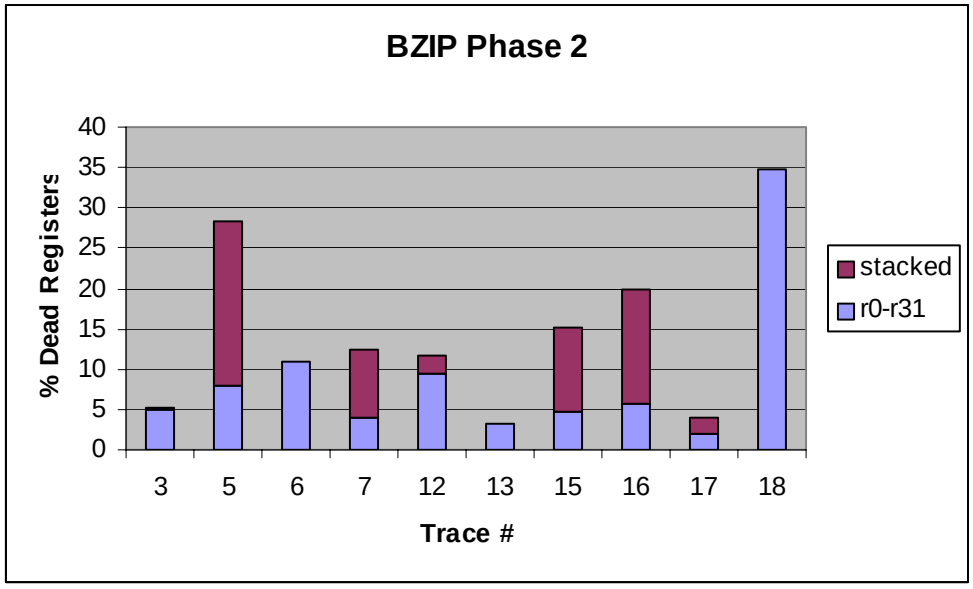
Trace #	1	2	3
# Instructions	13	9	8
# Registers	8	4	9



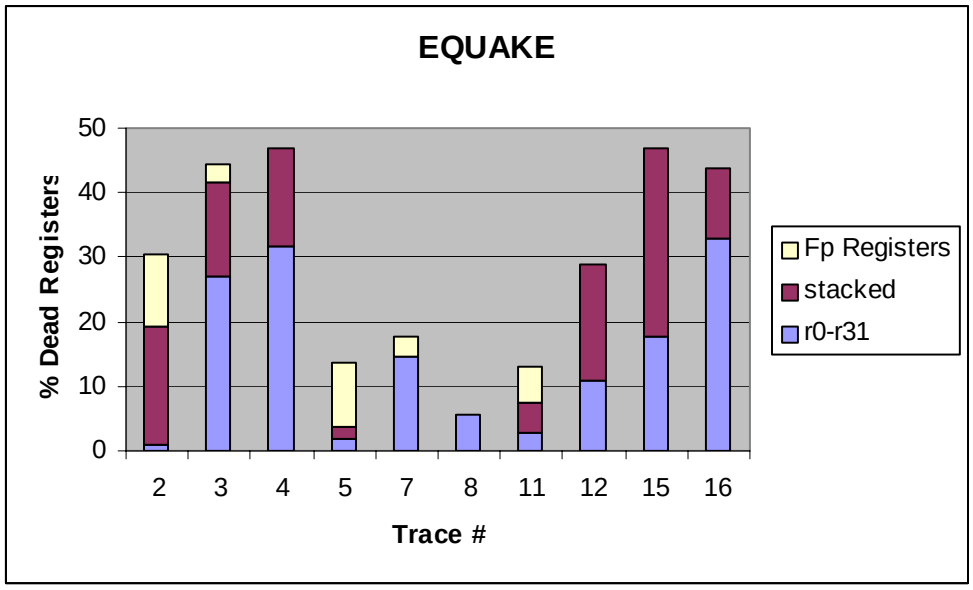
Trace #	1	2	7	8	9
# Instructions	188	371	7	8	11
# Registers	41	36	5	5	6



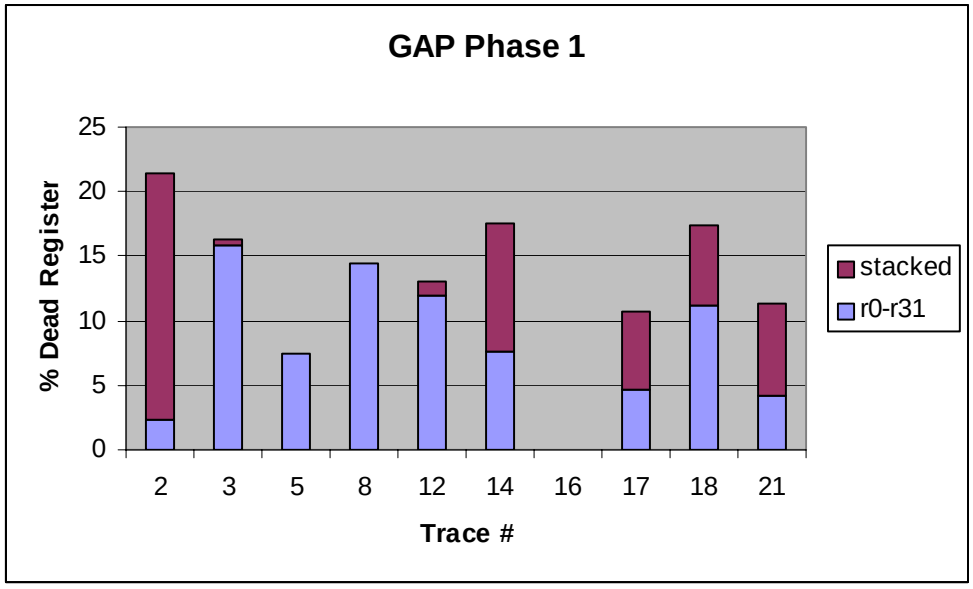
Trace #	3	4	7	8	9	12	13	18	20	21
# Instructions	224	10	14	15	45	23	17	18	18	36
# Registers	43	6	8	8	18	15	11	10	11	16



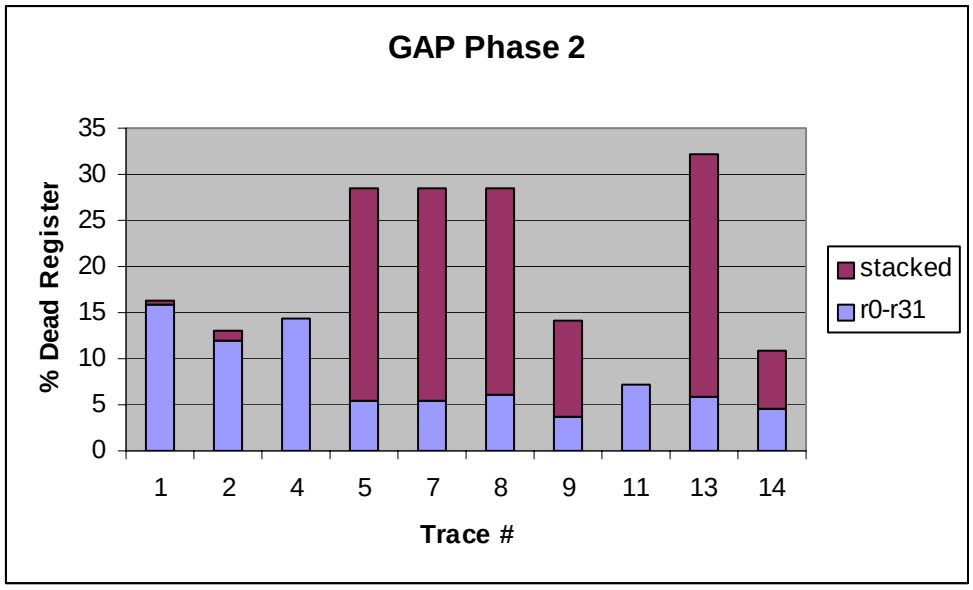
Trace #	3	5	6	7	12	13	15	16	17	18
# Instructions	70	25	15	14	24	10	18	31	10	9
# Registers	23	11	14	11	16	9	14	14	5	8



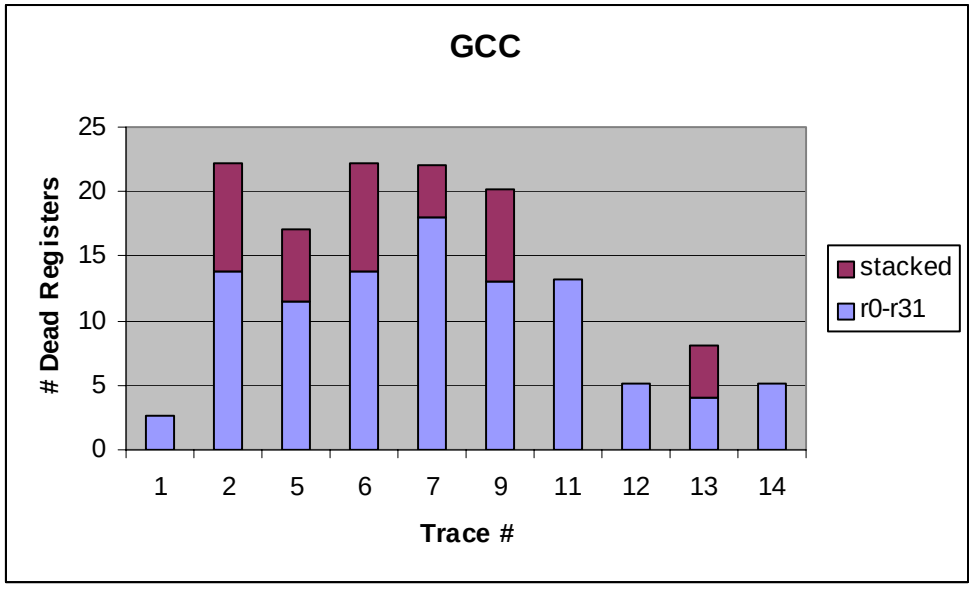
Trace #	2	3	4	5	7	8	11	12	15	16
# Instructions	63	58	24	30	30	9	18	66	30	24
# Registers	17	18	8	7	12	6	6	29	11	8



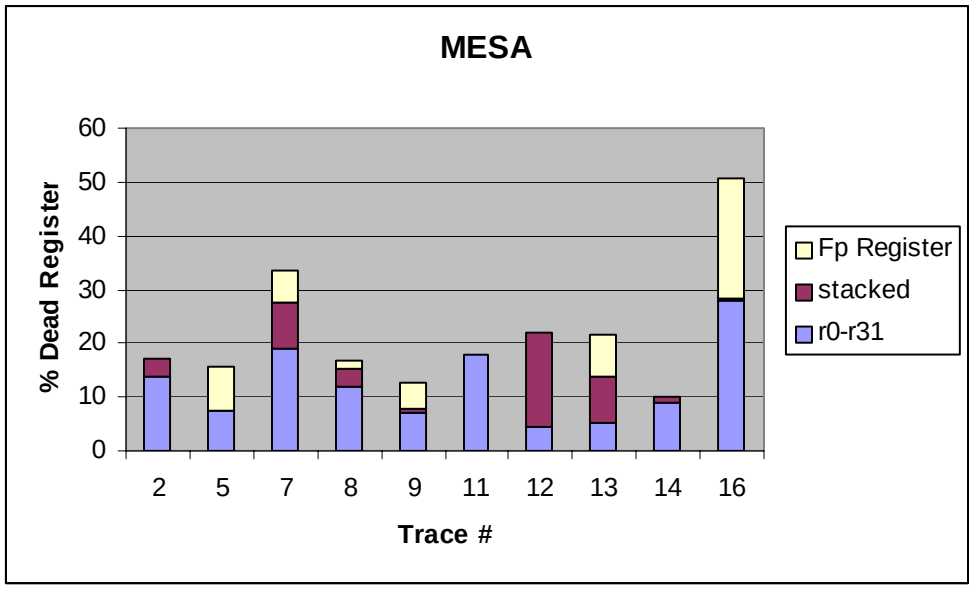
Trace #	2	3	5	8	12	14	16	17	18	21
# Instructions	7	86	19	43	121	19	4	24	9	53
# Registers	6	37	7	14	39	9	3	17	7	20



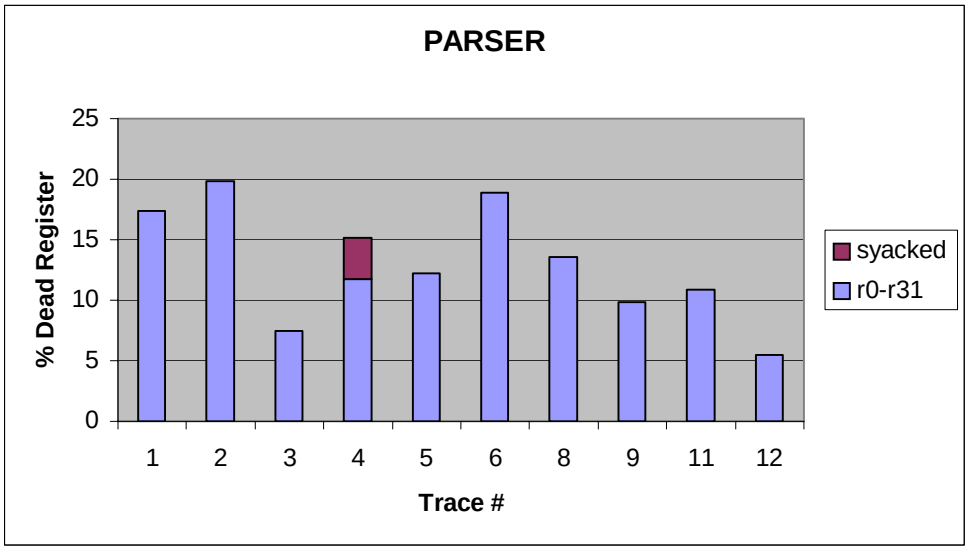
Trace #	1	2	4	5	7	8	9	11	13	14
# Instructions	86	121	7	13	13	13	16	20	17	24
# Registers	37	39	5	10	10	10	15	7	11	17



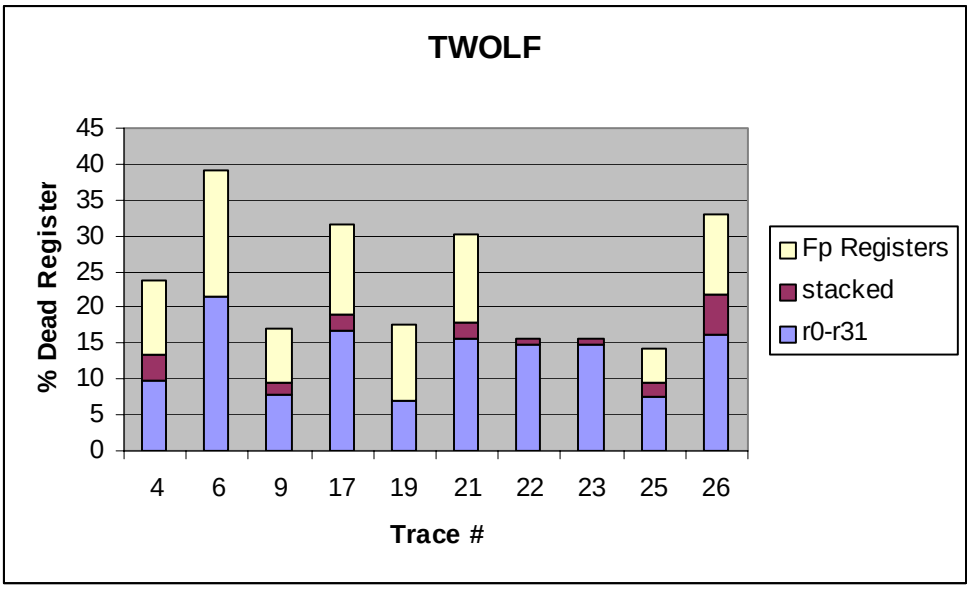
Trace #	1	2	5	6	7	9	11	12	13	14
# Instructions	26	9	14	9	10	17	12	39	18	34
# Registers	16	4	5	4	5	9	12	18	11	17



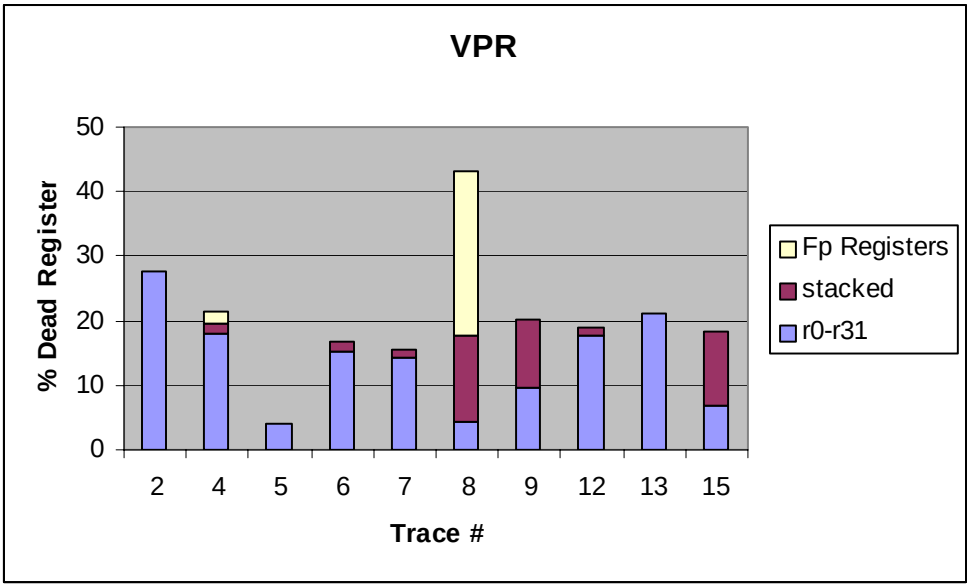
Trace #	2	5	7	8	9	11	12	13	14	16
# Instructions	81	24	111	63	116	33	12	173	23	29
# Registers	43	6	47	18	67	20	11	66	17	29



Trace #	1	2	3	4	5	6	8	9	11	12
# Instructions	93	16	15	51	70	13	21	27	70	35
# Registers	16	11	8	24	18	11	13	12	19	12



Trace #	4	6	9	17	19	21	22	23	25	26
# Instructions	96	42	197	28	28	60	34	34	181	121
# Registers	40	13	47	14	14	22	19	19	44	43



Trace #	2	4	5	6	7	8	9	12	13	15
# Instructions	27	18	60	23	55	32	14	34	21	13
# Registers	13	8	14	14	17	17	11	19	8	10

2.2 Register Spilling

Register spilling refers to storing the value of a register to some memory location upon entrance to the trace. Only those registers, which are not present in the trace, are likely candidates for spilling, since otherwise the overhead involved in spilling and restoring live registers would be high. The register value is restored to the original value upon exit from trace. This need to save and restore the registers upon trace entry and exit introduce extra overhead. There are several issues related to register spilling.

1) *The memory location used for spilling*: This seemingly trivial problem occurs because to store a register value to memory, another register is required to hold the address of the location, which is not available. User memory stack is a likely candidate for storage. No extra register is required because the stack register is already present. However, this may produce incorrect execution of the program if the code is written to use space below the stack pointer. Such a situation is likely to arise only in case of hand written assembly code. Itanium runtime convention guide advises applications against writing below the stack pointer [10].

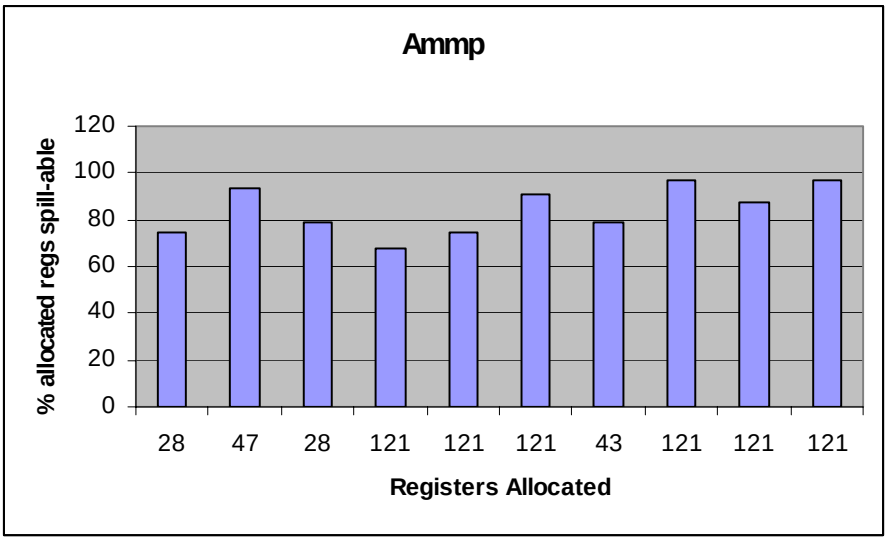
2) *Asynchronous Exceptions*: Exceptions prohibit using the stack as spilling area because during exception processing, activation record of the signal handler is built on top of the stack frame of the currently executing function. This can overwrite the spilled register values. Also as in the case of dead registers, there is a possibility of a global variable allocated in a register being accessed during handler execution.

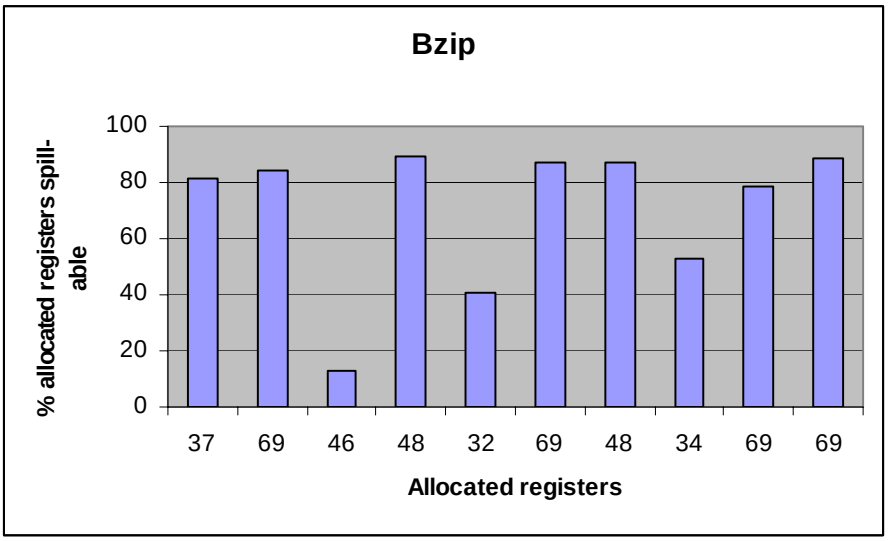
3) *Candidate registers to spill*: We said earlier that registers not present in the trace are likely candidates for spilling. However, just by examining the trace it is not possible to determine the number of registers that have been allocated by the procedure, and hence the registers not present in the trace. Hence we need to know exactly the size of the

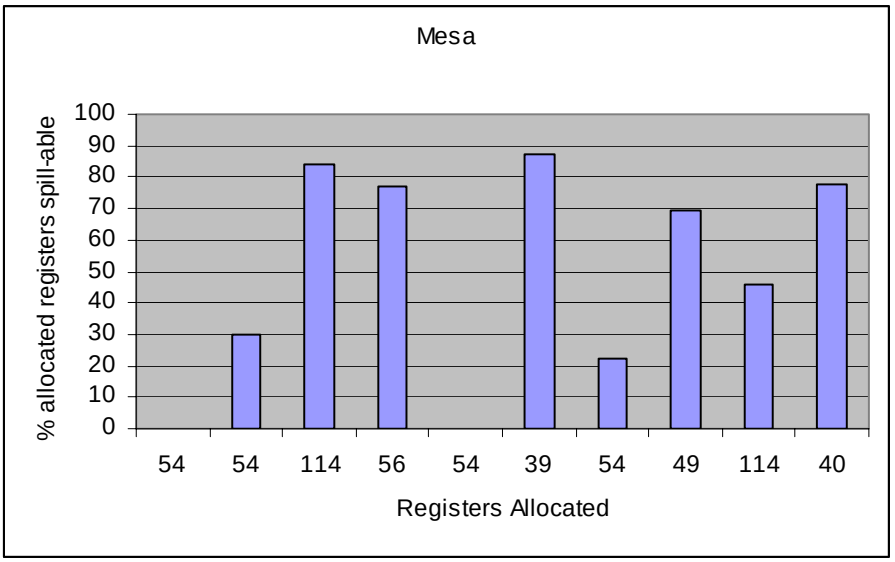
stacked register frame. A technique that has been used to allocate extra registers using the *alloc* instruction can be used for this purpose. The function is traversed backwards from the beginning of the trace to search for the *alloc* instruction, which is the first instruction executed upon function entry. The register stack frame size can then be determined by decoding the instruction opcode. This technique however, might fail in case there are multiple *alloc*'s used in the function.

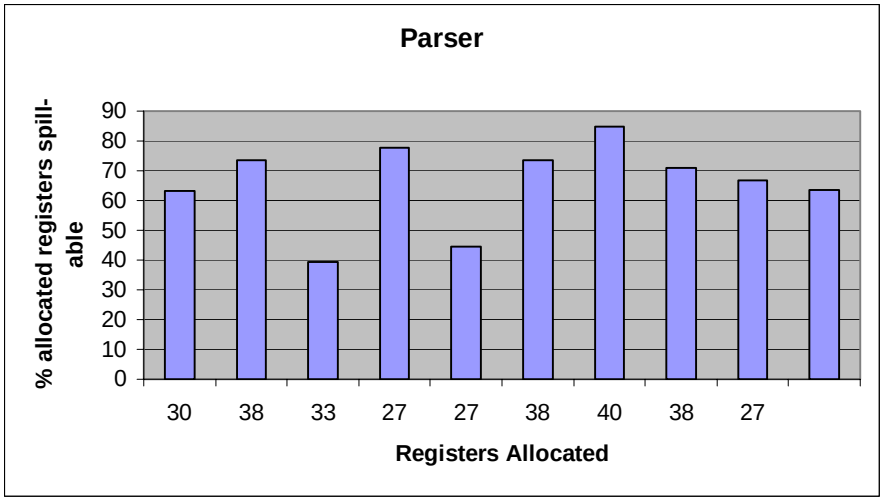
The graphs on the next page show data on the percentage of registers that can be spilled in a trace as a percentage of the total number of registers present in the function, for top ten traces of SPEC2000 benchmarks. The X-axis shows the number of registers allocated by a procedure and the Y-axis shows the percentage that can be spilled. The binaries are O2 compiled with ORCC.

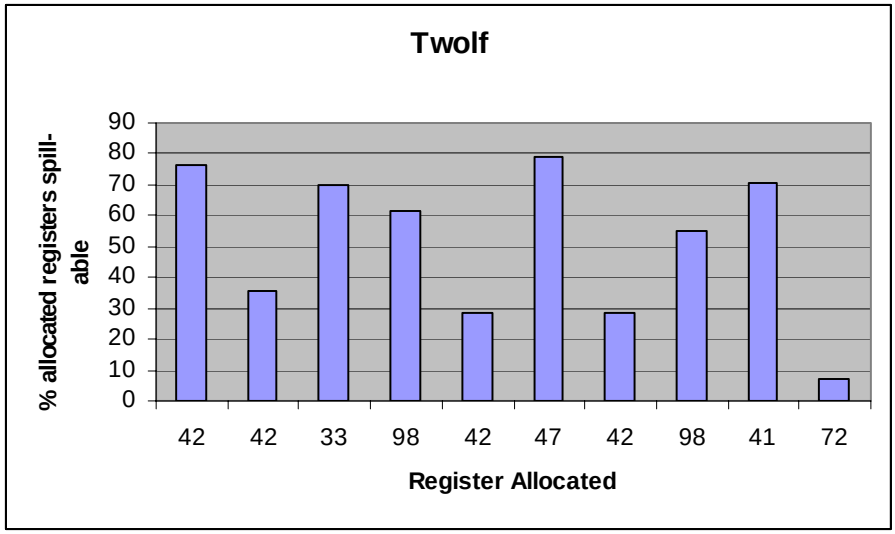
Following registers have not been considered for spilling: R0 (constant), R1 (global data pointer), R8-R11 (return value registers), R12 (memory stack pointer) and R13 (thread pointer) [10]. If they are present in the trace, we ignore them. The spill-able registers have been calculated by subtracting the number of registers present in a trace from the total number of registers allocated by the function. This includes the static registers and the stacked registers. Whenever a call is found in a trace, we subtract the *sol* field instead of the *sof* field. This is because output registers if spilled would need to be loaded before the call, increasing the overhead. We find that possibly a large number of registers are available for spilling. We see that for Ammp, ~65-98% of the allocated registers in a function can be spilled in a trace. The ORCC compiler does not seem to follow the convention of only eight registers in the output region. Sometimes it is allocating much more. Thus we see that in some traces of mesa, there are no registers to be spilled. This happens because there is a call in the trace and the number of output registers is large.

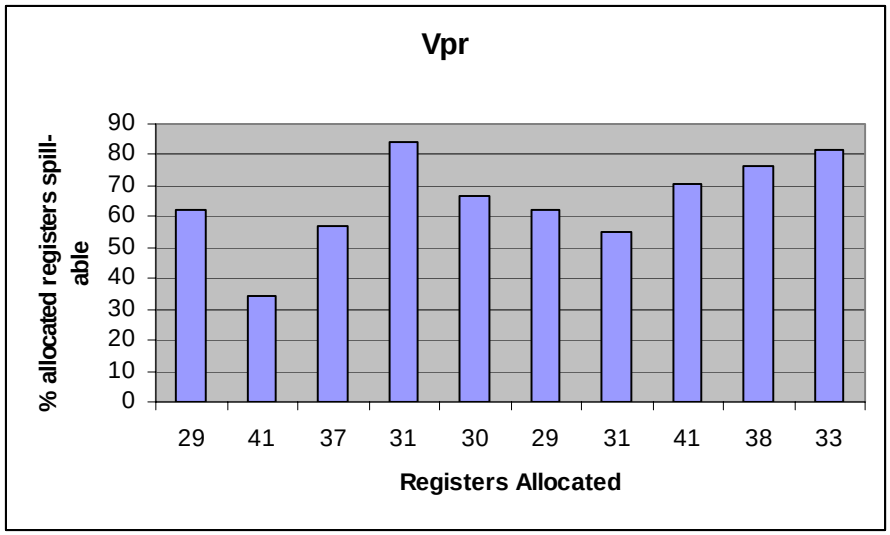

Ammp
% allocated regs spill-able
120
100
80
60
40
20
0
28
47
28
121
121
121
43
121
121
121
Registers Allocated

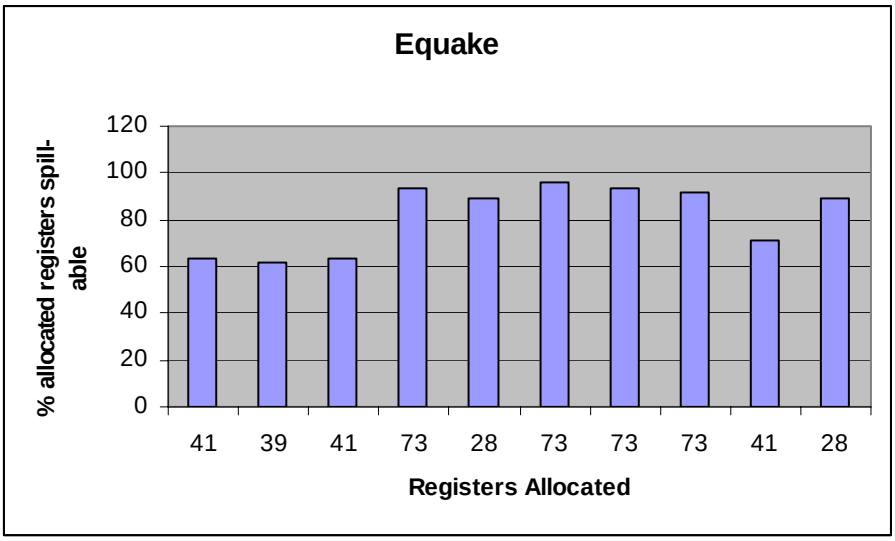

Bzip
% allocated registers spill-able
100
80
60
40
20
0
37
69
46
48
32
69
48
34
69
69
Allocated registers

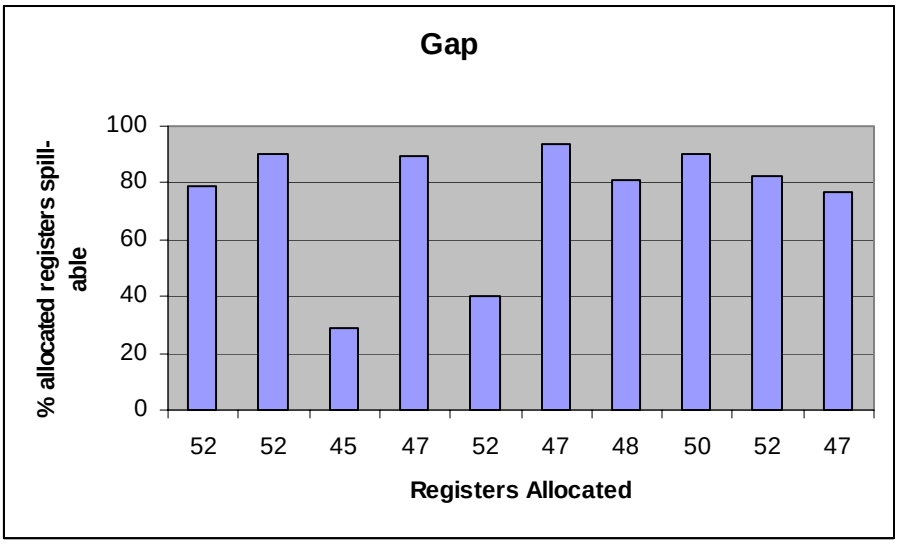

Mesa
% allocated registers spill-able
100
90
80
70
60
50
40
30
20
10
0
54
54
114
56
54
39
54
49
114
40
Registers Allocated

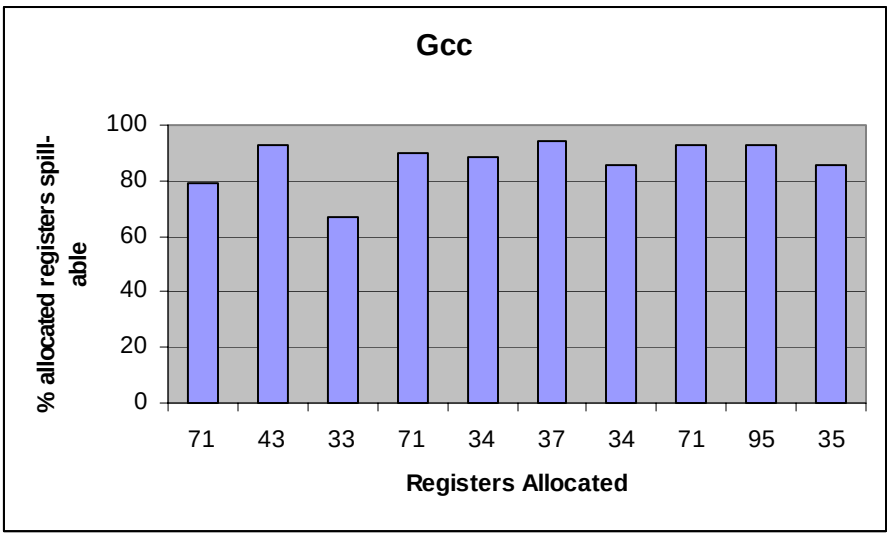

Parser
% allocated registers spill-able
90
80
70
60
50
40
30
20
10
0
30 38 33 27 27 38 40 38 27
Registers Allocated


Twolf
% allocated registers spill-able
90
80
70
60
50
40
30
20
10
0
42 42 33 98 42 47 42 98 41 72
Register Allocated


Vpr
% allocated registers spill-able
90
80
70
60
50
40
30
20
10
0
29 41 37 31 30 29 31 41 38 33
Registers Allocated


Equake
% allocated registers spill-able
120
100
80
60
40
20
0
41 39 41 73 28 73 73 73 41 28
Registers Allocated


Gap
% allocated registers spill-able
100
80
60
40
20
0
52 52 45 47 52 47 48 50 52 47
Registers Allocated


Gcc
% allocated registers spill-able
100
80
60
40
20
0
71 43 33 71 34 37 34 71 95 35
Registers Allocated

2.3 Using *alloc* Instruction

IA-64 architecture provides *alloc* instruction through which, a new stack frame is allocated on the general register stack [5], [7]. This avoids unnecessary spilling and filling of registers at procedure call and return through register renaming, which is under the control of the compiler. The virtual register identifiers in the instructions are renamed through a base register into the physical registers. The *callee* can freely use the available registers without having to spill and later restore the *caller*s registers. The *alloc* instruction specifies the number of registers the *callee* expects to use. If sufficient registers are not available, the instruction stalls the processor and spills the *caller*s registers into a backing store memory. During return from a function, the base register is restored to the *caller*s value prior to the call. If the registers have been spilled by the hardware and not yet restored, then the instruction stalls until processor has restored the registers. General register r32 to r127 are stacked registers in the architecture.

2.3.1 The *alloc* instruction [7]

The assembler format of the *alloc* instruction is shown below.

(qp) alloc r1 = ar.pfs,i,l,o,r

This instruction can be used to increase or decrease the size of the current register stack frame by changing the parameters *i, l, o,* and *r*. The previous function state register *ar.pfs* is copied to register *r1*. The change of the frame size is immediate with register *r1* belonging to the new frame. The parameters *i*(size of inputs), *l*(size of locals), *o*(size of outputs), and *r*(size of rotating) specify the size of the stack frame. Figure 2 shows the allocated frame and the fields.

2.3.2 Using *alloc* for Dynamic Allocation of Registers

The *alloc* instruction can be used to allocate extra register in the output area beyond the

current stack frame. The instruction is simply executed upon entry to the trace so that the

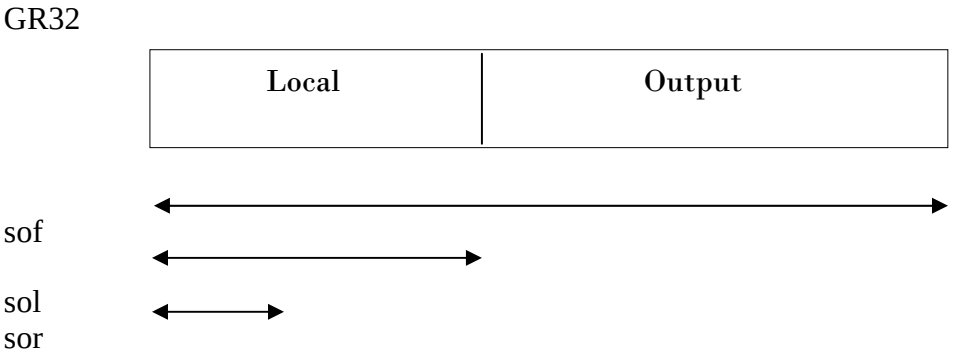


Figure 2: Register Stack Frame of a procedure.

output area has more registers. The scheme is attractive because potentially a fair amount of extra registers can become available by executing a single instruction. This scheme has been successfully used in the current implementation of ADORE system.

2.3.3 Issues

There are several issues and problems with using *alloc*, which may prohibit its use in some situations.

1) The current register usage of the procedure may extend almost to the maximum available stacked registers. Thus we may have very few or none of the registers available in the stack to bump.

2) The use of *alloc* to increase the frame size may cause Register Stack Engine (RSE) stalls because extra registers may not be available and the processor may have to stall execution until it spills the dirty registers to the backing store. This situation is unpredictable and out of control of the software. The time spent in spilling may amortize the benefits of any possible gain through optimizations. Figure 3 shows the overhead of using alloc for register allocation in ADORE for some SPEC2000 benchmarks. The overhead is the time required to scan for the *alloc* and executing the instruction.

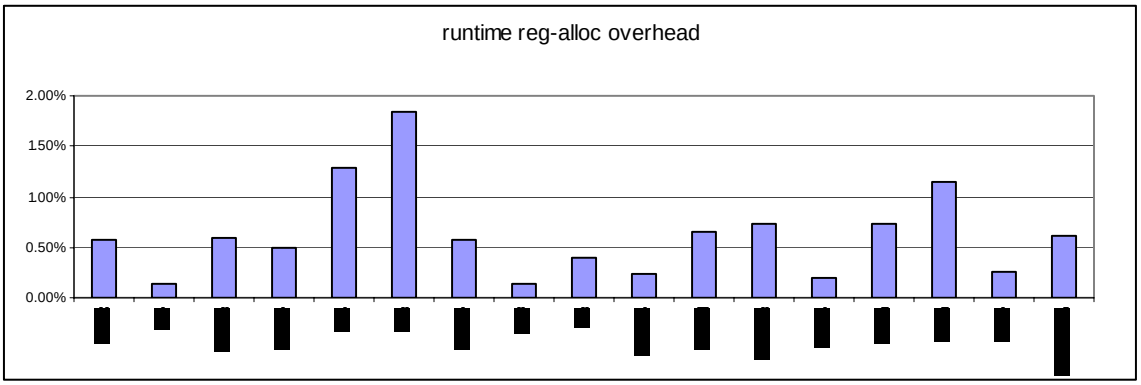


Figure 3: Overhead of using alloc for register allocation.

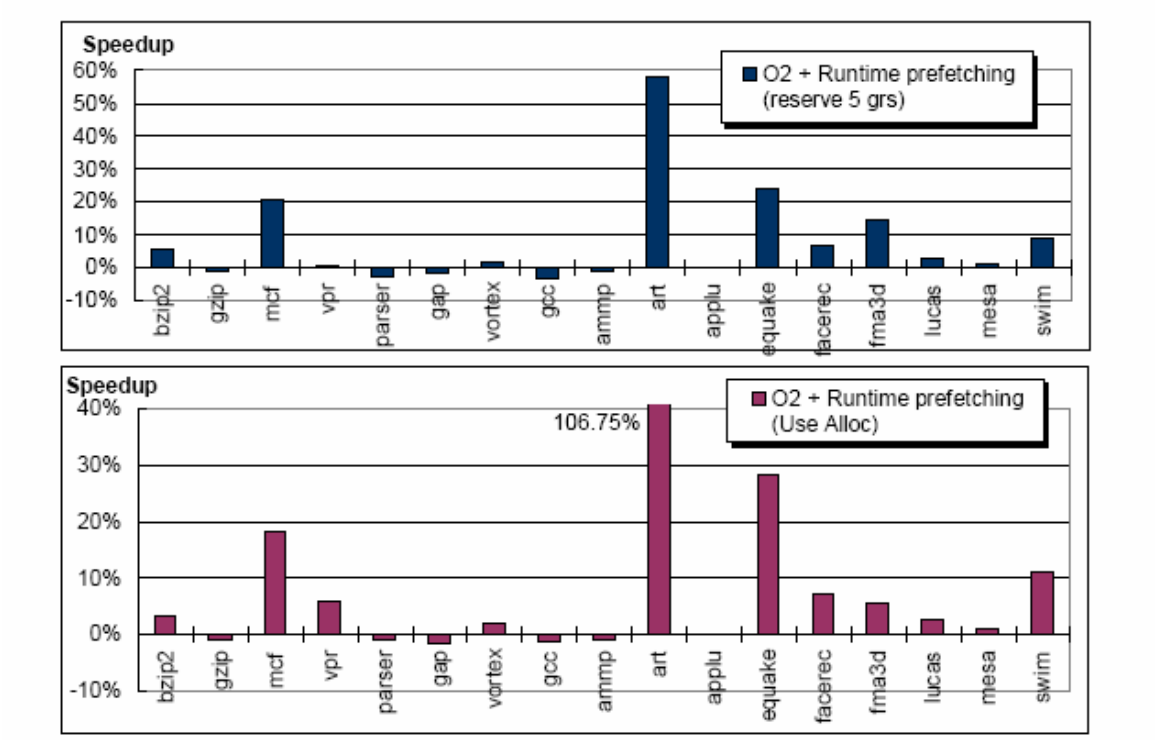


Figure 4: Speedup comparison in case of statically reserved registers and using alloc to allocate registers dynamically

Figure 4 [1] however, suggests that substantial speedups can be obtained when more registers using *alloc* are available, as compared to the case when only five registers are reserved statically at compile time.

3) In case there is a procedure call in the trace then the use of registers in the output region may be prohibited. This is because the output region of the *caller* is visible to the

callee as the input registers. Even though we are careful not to touch the registers used to pass parameters, ensuring correctness, the extra output registers can be used as scratch registers by the *callee*. This may destroy the values we may have stored for possible optimization. At the least, we may loose the advantage of intended optimization, if it was architecturally safe. At worst, we may end up with incorrect execution for more ambitious optimizations.

4) To increase the frame size, knowledge about the original stack configuration is required, i.e. we need to know the current frame size and the size of the output region since we allocate the registers beyond the output region of current stack frame. This information can be obtained by examining the encoding of the *alloc* instruction, which specifies the various regions of the stack. One approach is to trace backwards, from the location where we intend to place the *alloc*, to the function beginning in the code. Generally the *alloc* instruction is the first instruction executed by any procedure requiring a stack frame. The instruction can be decoded to determine the frame size and the second *alloc* coded accordingly. This approach though simple and applicable to most of the cases, is not general. The following cases illustrate when the approach may fail.

(i) The procedure body may have multiple *allocs* controlling different execution paths. This may be done by the compiler to reduce register pressure, which indirectly affects RSE stalls. One such scenario is shown in the control flow graph of figure 5. There are two paths of execution controlled by an IF-ELSE construct. If the IF path is the more frequently executed path requiring lesser registers than the seldom executed ELSE path, the compiler may use two *alloc* instructions allocating different number of registers along different paths.

In such a scenario the simple approach of traversing back the instruction sequence may

not work because it is impossible to statically determine, which *alloc* is controlling which execution path.

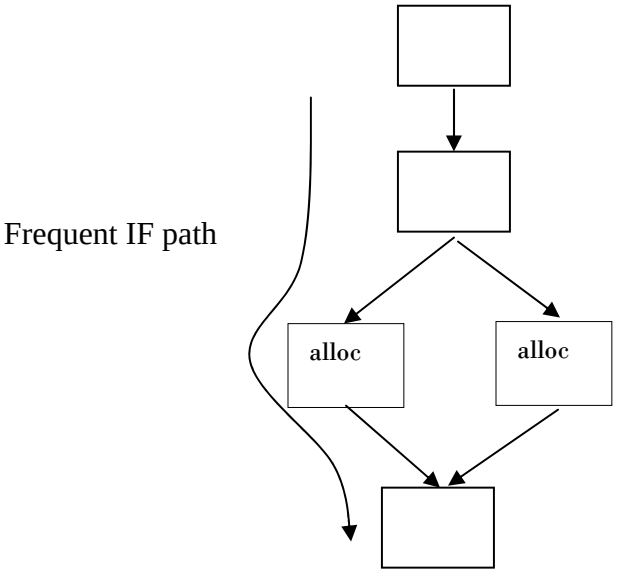


Figure 5: Code fragment with multiple allocs on different execution paths.

(ii) Certain static profile based compiler optimizations targeted towards instruction cache performance may not preserve the continuity of a procedure body. The procedure may be split into several parts based upon the frequency of execution and code from hot portions of other procedures may be inserted to maintain continuity of instruction stream. Examples of such optimizations are procedure splitting and hot cold optimizations [3], [4]. If the procedure body is split then traversing back to find *alloc* may no longer be feasible.

However some workarounds for this situation are possible. In the next section we discuss techniques that could be used to bump up the register stack indirectly, i.e. without the use of *alloc*.

2.4 Manipulating Register Stack Engine (RSE) for register allocation

In this section we describe two techniques that manipulate the internal state of RSE to bump the register stack. The techniques are dependent upon the processor architecture. We first describe the RSE and its internal state, followed by a discussion of the techniques.

2.4.1 The Register Stack Engine (RSE) [6]

The RSE is responsible for moving registers between the register stack and the backing store in memory without explicit program control. It operates concurrently with the processor and tries to take advantage of the unused memory bandwidth by dynamically issuing register spill and fill operations. The operation of the RSE is specified by the Register Stack Configuration register (RSC). The register stack frames are mapped onto a set of physical registers. These registers operate as a circular list. Each register stack frame is associated with the Current Frame Marker (CFM) register [5]. It holds the state of the current stack frame. The CFM is not architecturally accessible. The organization of the CFM is shown in table 3 below.

Table 3: CFM organization

Field	**Bit Range**	**Description**
Sof	6:0	Size of stack frame
Sol	13:7	Size of local portion of the frame
Sor	17:14	Size of rotating portion of stack frame
Rrb.gr	24:18	Register rename base for general registers
Rrb.fr	31:25	Register rename base for floating point registers
Rrb.pr	37:32	Register rename base for predicate registers

We basically seek to change the value of *sof* in this register to effect the change in frame size. The value of CFM is copied to Previous Function State (PFS) register during a function call and to the Interruption Function State register (IFS) when interrupts occur. They are used to restore the state of CFM on returns from functions and interrupts.

2.4.2 Backing store memory and RSE internal state [6]

As shown in the figure 6, the backing store is organized as a stack in memory that grows from lower to higher memory address. The memory allocated is generally part of the paged virtual address space of the process. This necessitates a switch of backing store during each context switch. Two application registers store information about the backing store memory. These are, the Backing Store Pointer (BSP), which contains the address of the first memory location reserved for spilling the current frame (i.e. the location for GR32) and the BSPSTORE, which contains the address where the next RSE spill will occur. Only BSPSTORE can be read or written directly. BSP can only be read and is altered as a side effect of writes to BSPSTORE. All RSE activity occurs below BSP since it operates on current procedures parent's frames. The RSE maintains internal state elements that are not exposed to the programmer directly but are manipulated indirectly as side effects to instructions that deal with RSE. They store the physical register numbers in the register file that directly correspond to the BSP and BSPSTORE pointers in memory. Figure 6 shows this correspondence. It also shows various partitions of the register file and the instructions that manipulate them. The states beginning with RSE are the internal states. For a detailed discussion see chapter 6 of Itanium System Architecture Guide [6]

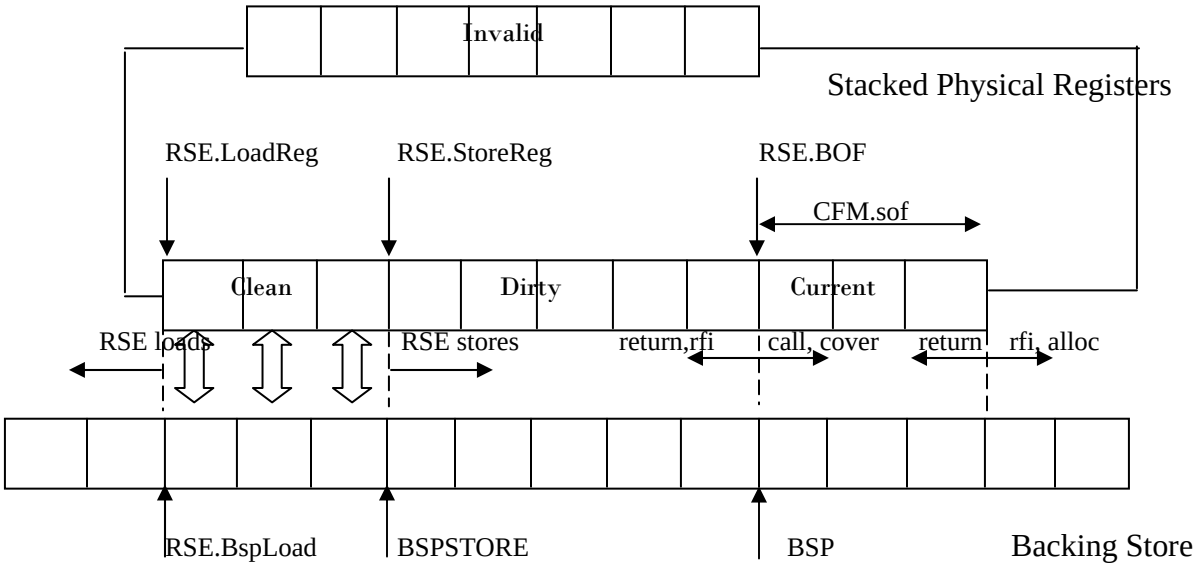


Figure 6: RSE and backing store organization

2.4.3 The approach used

During a context switch, the current register stack configuration of a process is saved in the user area associated with the process in the kernel address space. If we could somehow manipulate the registers that save information used by the processor to restore the stack configuration on a process switch-in, then we can increase/decrease the available registers at will. The reason this scheme can overcome the limitations listed in 4) of last section, is that only the current stack configuration in use at the time of switch out would be saved. Hence it does not matter how many *alloc*s were present in the actual function body. In the following subsections we describe the basic *ptrace* mechanism, followed by pertinent events that take place during a context switch, how the scheme works and issues involved.

2.4.3.1 *Ptrace() [9]*

Ptrace() is a system call available on Unix/Linux systems that allows a process to control the execution of another process. It is extensively used to implement break point debugging. The tracing process can attach to a running process or a child process can arrange to be traced by the parent process. For optimization purpose the tracing process could be the process responsible for deploying the optimization or a completely different process. The traced process runs normally until it receives a signal, when it is stopped and the tracing process is notified through the *wait()* system call. *Ptrace* then allows the tracing process to inspect and change the core image of the traced process. It also gives access to the set of registers saved in the user area of the traced process during switch out. These registers are accessed as offsets from the beginning of a kernel structure and can be found in a header file (ptrace_offsets.h for Linux).

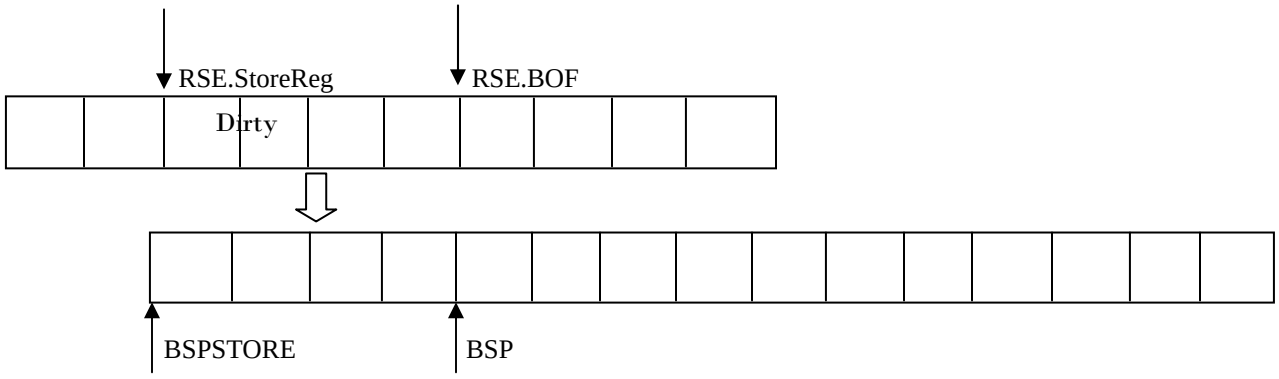
2.4.3.2 Backing Store Switch on Interruption [6]

The backing store is allocated in the virtual address space of a process. During a switch from user to kernel mode, the user backing store memory needs to be switched to kernel backing store and vice versa for switch from kernel to user mode. During a switch-out the following steps occur.

1. RSC and PFS registers are saved.
2. A cover instruction is issued which increments BSP to include the current frame. All registers between BSP and BSPSTORE are dirty, i.e. not yet saved to the backing store.
3. The CFM is copied to the IFS register on interruption. This register is also saved.
4. The BSPSTORE register is saved and is overwritten with the value of new backing store address. The BSP is updated to the value of BSPSTORE+ndirty,

where *ndirty* is the number of dirty registers. BSP is also saved.

The kernel stack frame looks as shown in figure 7, immediately after switch-out, and just before switch-in. The dirty (unsaved) registers are saved on the kernel-backing store during normal RSE operation.



7: State of register stack and kernel backing store just after/before process switch in/out.

2.4.3.3 Methodology

From the sequence described in the last section we can determine how the registers need to be changed to affect the increase in frame size upon return to user mode.

1. The value of CFM is stored in IFS register. So a change in *sof* field of IFS is required. To increase the frame size by 'n' registers, we add n to IFS. The *sof* field is the least significant 7 bits for IFS. It needs to be checked before incrementing so that it doesn't exceed the total number of physical stacked registers.
2. The value of PFS is changed to that of new IFS.
3. BSP is updated to (BSP+n*8) (each register is 64 bits). The change in BSP is required because the kernel restores the user mode registers using *loadrs* instruction, which loads the memory locations between BSP and BSPSTORE to the stacked register file and marks them as dirty. The processor then determines the physical register number of GR32 (i.e. RSE.BOF) by subtracting *sof* from the

physical register number, which corresponds to the value loaded from BSP. If BSP is not changed, and *sof* is increased then the register mapping would change, i.e. stacked register R, before changing the CFM, would be mapped to register R+n. To avoid this we increment BSP accordingly, allowing junk values from kernel backing store to be copied to the new output registers.

The only missing piece is how to transfer the control from user to kernel mode from the desired location. This can be achieved by making the process execute a break instruction with an immediate value of 0x80100. This break value is interpreted by the Linux kernel as a debugging trap. Signal SIGTRAP is sent to the executing process, which is stopped immediately.

2.4.3.4 Cost of using *ptrace*

Using *ptrace* to bump up the output region is not a cheap solution. At least six system calls are required to achieve this increase. Four calls for reading and setting the values of IFS and BSP, one for setting PFS and one to restart the process. Context switches are expensive operations and may amortize any benefit that could be gained from optimizations. Hence the optimizer must have strong reasons to apply this approach.

2.4.4 Using a function call and return sequence

The previous approach discussed makes use of IFS to change CFM. This approach uses PFS register to change the value of CFM. The scheme works by inserting a function call at that location in the code sequence, where we need extra registers. The called function executes *alloc* instruction to allocate its stack frame. The value of PFS is saved in a register. The *callee* then reads and updates the value of *caller*s PFS and executes a return. On return CFM is restored from PFS thereby increasing or decreasing the stack size. However, there is a problem with this approach. The RSE can detect a condition of bad

PFS when the following condition holds true

$$PFS.sof\text{-}PFS.sol > N\_STACKED\_PHYS - RSE.ndirty$$

i.e. the number of output registers specified is greater than total physical registers minus the registers in dirty partition. When RSE detects this situation it does not restores CFM to the value specified by PFS. Instead all the fields are reset to 0. To avoid this situation, a *flushrs* instruction can be executed before the return. This instruction flushes all the dirty registers to backing store, making *ndirty* zero. Hence the condition is avoided. However flushing registers stalls the process until the operation completes. Hence this might be an expensive operation depending upon the size of dirty partition.

2.4.5 Issues

With both the discussed approaches, the optimizer cannot know which registers to use for optimization until after it has created the trace and it has executed the break instruction or called the function responsible for incrementing the stack size. This implies that the optimization code has to be inserted after the trace has started execution. This is in contrast to approaches discussed earlier, when the optimizer could insert the code in traces while being created. The reason of course is that the stack configuration in which the trace is executing cannot be known till it starts execution. This increases the complexity of optimizer and places more stringent timing constraints on it.

3. Conclusion and future work

In this report we have discussed several techniques that can be used for dynamically allocating registers in a trace on Itanium architecture. Dead registers and register spilling are general approaches, applicable to different processor architectures. However the implementation is very ISA dependent. We have presented issues related to detecting and using such registers under the constraints of ISA, system and the source language. We also present experimental data on average lifetime of dead registers on several SPEC2000 benchmarks. We find that registers are dead from 0-70% of the total lifetime in the traces. We also present data on the number of registers that can be spilled in traces against total registers allocated in the function. Register spilling has a good potential to provide the needed registers for runtime optimizations. However, it remains a difficult issue to locate space for spilling registers. The cost of register spilling is also a concern for non-loop traces. This leads us to discuss the approach specific to Itanium architecture that uses *alloc* instruction to allocate extra registers. This scheme is attractive but fails in case of multiple *alloc*s. Two different approaches to overcome the limitation have been proposed. The schemes have been verified experimentally for correctness.

Future work includes implementing the schemes and including them with the optimizer. The effectiveness of various methods and overhead introduced for allocation can be quantitatively analyzed. An algorithm that can dynamically decide the most cost effective way to allocate registers is an interesting area that could be explored.

4. References

1. J. Lu, H. Chen, P.C. Yew, W.C. Hsu. Design and Implementation of a Lightweight Dynamic Optimization System. In *The Journal of Instruction-Level Parallelism,* vol.6, 2004.

2. Fisher, J.A. Trace Scheduling: A technique for global code microcode compaction. In *IEEE Transactions on Computers* 30:7 July, 478 – 490.

3. Karl Pettis, Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN* 1990.

4. Robert Cohn, P. Geoffrey Lowney. Hot Cold Optimization of Large Windows/NT Applications. In *MICRO29, Paris, France* (December 1996): 80 - 89.

5. Intel, *Intel IA-64 Architecture Software Developer's Manual*, Vol. 1: IA-64 Application Architecture.

6. Intel, *Intel IA-64 Architecture Software Developer's Manual,* Vol. 2: IA-64 System Architecture.

7. Intel, *Intel IA-64 Architecture Software Developer's Manual,* Vol. 3: IA-64 Instruction Set Reference.

8. Intel, *Intel Itanium Processor Reference Manual,* Software Development and Optimization.

9. http://unixhelp.ed.ac.uk/CGI/man-cgi?ptrace+2. ptrace() manaul.

10. Intel, *Itanium Software Conventions and Runtime Architecture Guide.*