

Optimization for the Intel[®] Itanium[®] Architecture Register Stack

Alex Settle Daniel A. Connors
Department of Electrical and
Computer Engineering
University of Colorado at Boulder
{settle, dconnors}@colorado.edu

Gerolf Hoflehner Dan Lavery
Intel Corporation
Santa Clara, CA
{gerolf.f.hoflehner,daniel.m.lavery}@intel.com

Abstract

The Intel[®] Itanium[®] architecture contains a number of innovative compiler-controllable features designed to exploit instruction level parallelism. New code generation and optimization techniques are critical to the application of these features to improve processor performance. For instance, the Itanium[®] architecture provides a compiler-controllable virtual register stack to reduce the penalty of memory accesses associated with procedure calls. The Itanium[®] Register Stack Engine (RSE) transparently manages the register stack and saves and restores physical registers to and from memory as needed. Existing code generation techniques for the register stack aggressively allocate virtual registers without regard to the register pressure on different control-flow paths. As such, applications with large data sets may stress the RSE, and cause substantial execution delays due to the high number of register saves and restores. Since the Itanium[®] architecture is developed around Explicitly Parallel Instruction Computing (EPIC) concepts, solutions to increasing the register stack efficiency favor code generation techniques rather than hardware approaches.

1. Introduction

This paper presents a compiler optimization strategy, named multiple alloc, aimed at reducing memory latency associated with RSE transfers by generating multiple register allocation instructions within a subroutine. While existing schemes generate a single allocation of stacked registers for an entire subroutine, the new optimization technique utilizes register liveness analysis to fine-tune register allocation. The optimization was developed on the Intel[®] Itanium[®] production compiler and experimental results indicate that this optimization can expose significant savings in RSE register-memory traffic.

Currently memory performance issues are limiting the

performance of modern architecture implementations. Previous studies [4] report that nearly 50% of the processor execution time is spent stalled, waiting for both instruction and data cache misses. Explicitly Parallel Instruction Computing (EPIC) architectures allow the compiler to express program instruction level parallelism directly to the hardware to deal with increasing memory latencies and penalties. Specifically, the Itanium[®] architecture deploys a number of EPIC techniques which enable the compiler to represent control speculation, data dependence speculation, and predication [3] to enhance performance. These techniques have individually been shown to be very effective [2] in dealing with memory penalties. In addition to these techniques, the Itanium[®] architecture provides a virtual register stack to reduce the penalty of memory accesses associated with procedure calls and to leverage the performance advantages of a large register file. However, in order to make effective use of the register stack, significant advances in code generation and optimization techniques should be developed.

In the Register Stack (RS) model, a procedure maintains a register stack analogous to the memory stack [15]. Itanium[®] processors implement the register stack as a large physical register file with a hardware-controlled mapping between virtual and physical registers. The register stack model can significantly reduce the number of loads and stores required to save registers across procedure calls.

Architectures with a register stack typically allocate a single variable-sized register stack frame to each procedure. This paper uses the terms register stack frame and stack frame interchangeably. The register stack enables the compiler to analyze the register requirements for the entire procedure and to specify the register stack frame size with a special register stack management instruction. The architecture ensures that register stack frames for different procedures do not overlap. The RSE is a processor state machine defined by the architecture. It maps a register stack frame onto the physical register file and copies values in registers to and from memory in response to overflow and underflow

conditions respectively. Each program has its own backing store in memory. In the event of an overflow of the stacked register file, stack frames are copied, or spilled to memory, then retrieved or filled for underflow conditions.

This paper describes how to improve register stack performance through the use of careful code generation techniques. The current implementation of the RSE initiates spilling and filling of stacked registers after *alloc* and return instructions respectively [9]. RSE spilling and filling operations stall the pipeline in order to generate implicit stores and loads. Since traditional register stack code generation methods apply only single-allocation management techniques per procedure, a large number of registers may be redundant across call sites. This could result in a significant performance loss, as the RSE would spill and fill dead registers.

Rather than propose an alternative hardware design, this paper demonstrates a more efficient way to generate code for the Itanium[®] architecture's register stack. The compiler deploys new code generation and analysis techniques to find strategic program points and individual allocation amounts that can improve the efficiency of the RSE. The basis for the optimization is that the compiler can resize the register stack frame at any point within a procedure, thereby allowing fine-grain register stack allocation based on control flow, live-variable tracking, and procedure call analysis [8]. Overall, this paper illustrates a new compiler optimization method that can reduce the register-memory traffic from the physical register file to the memory system. Before explaining the details of the multiple *alloc* optimization, we describe the Itanium[®] Register Stack design.

1.1. Itanium Register Stack Background

The Itanium[®] architecture provides a general register file with 128 registers, r0 - r127. The register file is partitioned into 32 static registers, r0 - r31, and 96 stacked registers, r32-r127. The static registers are visible to any procedure, however, the stacked registers, with the exception of registers used for parameter passing, are only accessible locally within a procedure. Stacked registers are organized in units of stack frames. Each stack frame is associated with a particular function. The registers in a stack frame are preserved across call boundaries. On the Itanium[®] architecture, the register stack manager is called the Register Stack Engine (RSE) and the memory system is called the backing store [9].

On the Itanium[®] processor, each procedure can have its own variable size register stack frame with its own variable number of rotating registers. Stacked registers are grouped into four classes: rotating, input, local and outgoing. The input registers contain the parameters passed to the function, the locals are registers that the function will use dur-

ing its lifetime, and the outgoing registers are parameters that are passed to functions called by the current function. A procedure allocates a register stack frame by invoking the *alloc* instruction [17]. This instruction has the format [9]:

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

Figure 1 illustrates the manner in which virtual register names are mapped to the physical register file. In this example, the functions *foo()* and *bar()* both access stacked registers starting with r32, allowing the generated code to remain unaware of the actual physical registers. When the function *foo()* calls *bar()*, it passes arguments to *bar()* through the outgoing stacked registers. When *bar()* executes, it reads its arguments from the incoming stacked registers which actually refer to the same physical storage location as *foo()*'s outgoing registers (physical registers r39 and r40).

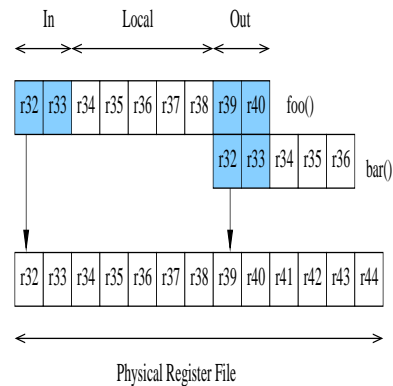


Figure 1. Alloc instruction and register stack frame mapping.

A register stack frame consists of the following register classes: a maximum of 8 incoming parameters (in); up to 8 outgoing parameters (out); the number of locally allocated stacked registers (local); and optionally the number of rotating registers (rot) used in software-pipelined loops. The total number of registers in the register stack frame for a procedure is $in+local+out \leq 96$.

1.2. RSE Virtual to Physical Mapping

The RSE is modeled upon the concept of a memory stack. The actual register stack is implemented as a circular buffer in the processor core. The circular register file illustrated in Figure 2 provides the abstraction of an infinite register file viewed across the lifetime of the process. The size of a register stack frame per function may be up to 96 registers. The maximum frame size is a hard limit imposed by the architecture, and for future Itanium[®] models the maximum frame size must remain 96 [9]. Nevertheless

the size of the stacked register file is dependent on the implementation of the architecture.

According to Figure 2, the Current Frame Marker (CFM), and the Top of Stack (TOS) pointer define the boundaries of the current active stack frame. The physical stacked registers are grouped into four categories: active, invalid, clean, and dirty. The active registers are those used by the current stack frame. The invalid are registers that are ready to be written. Clean registers have been spilled to memory and can be overwritten. Dirty registers contain values that must be stored to memory before they can be allocated to another frame. In this example, when the next function is called, the active set becomes dirty, the CFM moves to one beyond the TOS, and the new TOS is set by the application. Invalid registers are designated for use by the newly activated function. Overflow occurs when the invalid region is lost to active, dirty, and clean registers. When this occurs, stacked register frames from the dirty section must be stored in memory, making them clean, before they can be allocated to a new function.

When a call instruction is executed, the register stack frame of the caller is saved, the first outgoing register of the caller is renamed to r32 and the stack frame of the callee consists of the outgoing registers of the caller. With the *alloc* instruction the callee can then resize the new register stack frame [9]. Currently, the Intel[®] Itanium[®] compiler [9], usually issues one *alloc* instruction at each procedure entry. This instruction must allocate the maximum number of registers used on any path through the function.

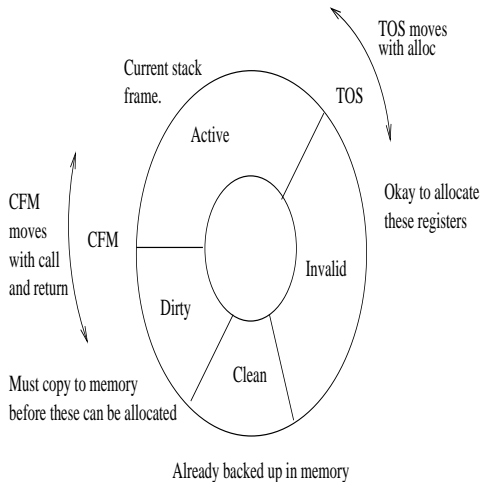


Figure 2. Circular buffer register mapping.

1.3. RSE Spilling and Filling

In the event of an RSE overflow, stacked registers must be spilled to memory until the physical register file has enough registers to accommodate a new register stack

frame. Spilling registers to memory causes the processor to stall until the spills complete. Spilling or filling has the potential to cause large performance penalties, since many stacked registers may be stored to or loaded from memory before the program can resume execution. Figure 2 demonstrates the different classifications of stacked integer registers from the perspective of the physical register file. Spilling occurs when the invalid region has fewer registers than needed by the current active function. The multiple allocation optimization attempts to reduce the spill delay by decreasing the size of a function’s stack frame across procedure calls where stack overflows may occur.

The *alloc* instruction determines the size of the of the register stack frame. A second *alloc* instruction in the same procedure does not cause RSE spills, when it allocates a smaller stack frame than the previous *alloc* instruction [9]. By carefully tracking the number of live registers at strategic locations in the control flow graph, additional *alloc* instructions can be inserted to modify the size of the current stack frame. Thus a multiple allocation optimization can exploit this property by reducing the stack frame size before a call site. This frees up registers which can be used in the stack frame for the callee and may reduce the overall register-memory traffic for an application. After returning from the function call, another *alloc* instruction may restore the original stack frame at no extra spill cost.

Figure 3 demonstrates the case of four interdependent procedures: A, B, C and D. Procedure A is called first, which then calls procedure B. Procedure C is called from B, and procedure C calls D. In this example, procedure C is the active stack frame, and it will call procedure D. In this case, there are not enough stacked registers available for function D, so the RSE must free stack space for D by spilling part of A’s stack frame. The frame allocated to procedure D consists of the remaining sixteen invalid registers, and the bottom fourteen registers which formed part of A’s stack frame. Finally, after procedure B returns, the RSE will fill the remainder of stack frame A back into the register file. If the multiple *alloc* optimization had been used at the call site of procedure D, then the number of spilled registers would have been reduced or eliminated entirely.

2. Related Work

There has been significant related work in the area of register allocation, but the RSE appears to be unique in that it provides autonomous fine-grained mapping of virtual to physical registers. The SPARC architecture uses register windows in a similar manner as the RSE. Register windows allow function parameters to be passed through designated registers and also provide a small set of registers for storage local to a procedure [18]. The interface to these registers is less configurable than the interface to the Itanium[®]

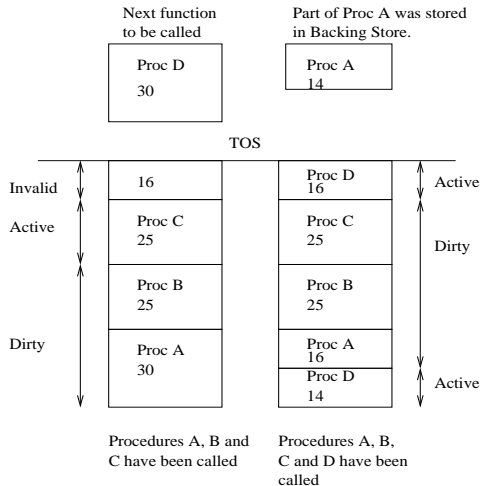


Figure 3. RSE Spill Example.

stacked registers. In particular, the register window size is fixed at twenty four registers, while an Itanium[®] register stack frame can be sized at run-time by an *alloc* instruction. Both the SPARC and the Itanium[®] overlap the outgoing parameters of the caller with the incoming parameters of the callee [11].

Other studies have been done on the issue of increasing the register file size. One such study [16] proposes a caching mechanism used to access the set of registers in a larger register file. The important problem to address is that with a larger register file access times increase, making it more difficult to access the register file in a single clock cycle. Providing a caching scheme or some other fast look-up procedure may make it possible to reap the rewards of a large register file without suffering a delay in access time [12].

Another compiler directed RSE optimization offers a brute force method of issuing the *alloc* instructions [6]. This approach analyzes stacked register usage per basic block, and issues the *alloc* instruction if the stack frame can be reduced for the basic block. The multiple alloc algorithm differs in that it takes a more global view of the control flow graph and issues *alloc* instructions more strategically.

The multiple alloc algorithm exploits the existing implementation of the RSE on the Itanium[®] Processor Family (IPF) architecture. Rather than examine run-time or profile guided feedback, this optimization takes a purely static approach. Previous studies on the RSE [19] have proposed modifications to the RSE itself to provide a more efficient means of stacked register allocation. The multiple alloc optimization is unique in that it collects stacked register information at compile time and performs the optimization without profile information. This optimization offers a clean solution to the RSE performance problem by clever use of the existing architecture features.

3. Motivation

Although the RSE provides a significant reduction in memory traffic for most cases, spill and fill operations still cause long execution delays. Current Itanium[®] compilers pay little attention to RSE management. In many cases, the stack frame sizes set at compile time are far larger than the actual number of stacked registers used by a given function at run time. Thus, the register stack is subject to internal fragmentation from either dead [14] or unused registers. Should a spill or fill occur for this case, the register stack engine would cause a delay in program execution due to storing these dead and unused registers.

The multiple allocation optimization is a compiler optimization to control the stack frame size within a procedure. This may reduce the overall program stall time incurred by RSE spills and fills. Since the *alloc* instruction resizes a register stack frame, it does not trigger RSE spills when it shrinks a frame. The tradeoff for this optimization is an increase in code size by additional *alloc* instructions.

3.1. Motivating Case

Figure 4 demonstrates a motivating example for the multiple alloc optimization. Upon entry into a hammock region of code the procedure allocates 30 stacked registers. Control flow splits after this point sending the program to execute either *foo()* or *bar()*. The basic block containing the call to *foo()* likely has a different number of live registers than the block which calls *bar()*. In this example, the stack frame can be reduced to 10 registers for function *foo()* and 20 for function *bar()*. After returning from these functions, the stack frame is restored by an additional *alloc* instruction.

3.2. Liveness Analysis

In order to determine the number of registers by which the current stack frame can be reduced, the algorithm queries a register live range database that is generated by the register allocator. Starting with the top allocated local register, the algorithm checks register liveness. If the top register is live, the search stops without reducing the stack frame size. If the top register is dead, the process is repeated for the next local register. The stack frame size can now be reduced by the number of dead local registers that were identified by this search. The fundamental disadvantage of this approach is that it depends upon the last local registers being dead. For example, if a function allocates registers r32-r60 and, at a call site, registers r41-r60 are dead, then the stack frame can be reduced by 20 registers. If however, register r60 is live, then the stack frame must be preserved.

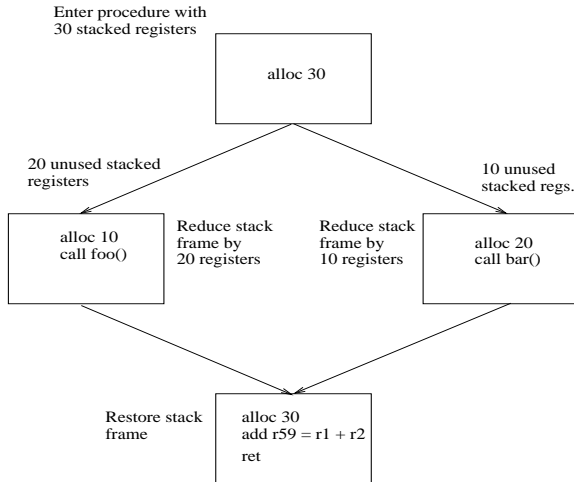


Figure 4. Control flow graph containing call sites with different allocation requirements.

This occurrence prevents the algorithm from issuing an *alloc* instruction, despite having potential for reducing internal fragmentation.

This shows that the multiple allocation optimization could be more effective when it can be integrated into the register allocator of the compiler. For example, liveness information could enable the register allocator to allocate live ranges crossing function calls to stacked registers at the bottom of the register stack frame. This could increase the number of dead registers at the top of the frame at a call site. Effectively the integrative approach would trigger changes in the coloring phase of a register allocation algorithm [5]. Similarly, region compilation techniques [7] could be deployed to effectively allocate stacked registers on a per region basis.

4. Approach

The multiple allocation algorithm was implemented in the Intel[®] Itanium[®] production compiler. It was designed as a default optimization level for the production compiler [3][13]. As such, many of the design considerations were made in order for it to work properly with the other Intel[®] Itanium[®] compiler optimizations. For example, copy propagation makes the implementation of the multiple allocation algorithm more difficult, because it allows for parameter registers to be defined outside of the call block. The problem of correctly identifying function parameter registers for each call site led to the development of a new data flow analysis algorithm based upon traditional reaching definition analysis [1]. Other optimizations, such as global

code motion and register coalescing, can also complicate parameter register identification.

4.1. Algorithm Example

The example control flow of Figure 6 helps explain the multiple allocation algorithm. In this particular case, each of the two control paths leads to call a block. Initially, r50 is the first outgoing argument, and is assigned a value in the predecessor to the call blocks. In single allocation mode, there is no problem posed by writing to a parameter register before the call block is reached. However, for the multiple allocation mode, the situation is more complicated. Since each call site can potentially change the number of local and outgoing stacked registers, it is important for the compiler to be able to associate parameter registers with the corresponding function calls.

4.1.1. Multiple Allocation at Call Site Locations

In the above example the algorithm, as shown in Figure 5, must first identify the two call blocks, then determine the number of stacked registers that can be saved at each one. If the number of saved registers falls below a certain threshold, then the routine ends and the control flow graph is unchanged. The threshold can be changed by the user in order to determine its optimal value. For the experiments described in this paper, the threshold was 4 registers. If the threshold is exceeded, then the next phase is to identify all of the parameter registers associated with the call. Locating all of a call site's parameter registers can be quite complicated when the parameter assignments are outside the call block.

The challenge with parameter assignments outside the call block is that it complicates, internally, the mapping of outgoing parameter registers to the corresponding call instructions. When the parameter assignments are local in the call block, then it is easy for the compiler to associate the outgoing parameters with a specific function call.

4.1.2. Reaching Analysis

Reaching analysis is activated in order to correctly identify the parameter registers for each call instruction. The algorithm can determine if any of these registers are shared with other calls. There are several actions in response to identifying a parameter register that is defined globally. The simplest action is to not issue the *alloc* instruction. This will preserve program correctness, but will abandon the benefits provided by the optimization. A better approach is to copy the contents of the global outgoing register to its new register location before issuing the *alloc* instruction. This will

```

input: optimized code
method:
  foreach callblock b
    if (NOT register_stack_frame_is_reducible)
      then CONTINUE;
    endif
    call = CALL_INSTRUCTION(b);
    additional_move = False;
    foreach param reaching call
      if (NOT assigned_in_this_block)
        additional_move = True;
      endif
    endforeach
    if (additional_move)
      new_parm_reg = old_parm_reg;
      INSERT_MOV_INSTRUCTION(b);
    endif
    INSERT_ALLOC_INSTRUCTIONS(b);
    REMAP_REGISTERS(b);
  endforeach
output: optimized code with multiple alloc instructions

```

Figure 5. Algorithm 1: Reducing register stack frames before calls.

ensure that the register is always contained within the current stack frame, and that the correct value gets passed to the call.

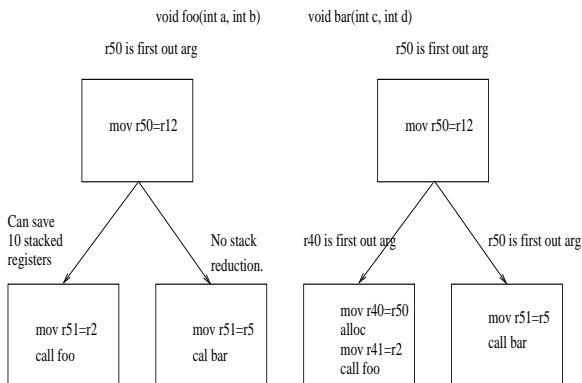


Figure 6. Copy Prop. Example

Figure 6 demonstrates the parameter register problem. For the control flow on the left, r50 is passed to both functions as the first outgoing parameter. Register r50 is also defined globally with respect to the two call blocks. If, for example, the compiler determines that 10 registers can be saved across the call to foo(), then it will issue an *alloc* instruction. If no registers can be saved across the function bar(), then the compiler must

be careful that both functions are passed the correct parameters. To do this for function foo(), r50 must be copied to r40 in foo()'s call block. This is done explicitly by inserting a *mov* instruction as shown by the pseudo code function INSERT_MOV_INSTRUCTION() of Figure 5. After issuing the new *alloc* instruction (INSERT_ALLOC_INSTRUCTIONS()), r40 becomes the first outgoing parameter register and r51 is no longer visible by this function. The second outgoing parameter, r41 is mapped automatically by the compiler function REMAP_REGISTERS() of Figure 5. Both functions now have copies of the correct parameter register values.

4.1.3. Algorithm Efficiency

The new algorithm is within the standard complexity of an optimizing compiler, and runs very fast (almost linear-time $O(N)$ in practice [15][1]. The method runs approximately in $O(\max(N, \text{\#call blocks} * \text{\#number of parameter references}))$. Similarly, the memory requirement for the algorithm is marginal. The reaching analysis associates five bit vectors with each basic block in the control flow graph. It uses a table to provide the mapping between bit positions and the corresponding instruction. All of these data structures grow linearly with respect to the number of parameter references in the procedure.

4.2. Register Stack Unwinding

Multiple *alloc* instructions within a procedure require no extra effort to support register stack unwinding. The register stack frame before a call defines the function state, which the callee must save in its unwind table. Unwinding simply restores the register stack frame before the call. As long as the system saves the previous function state at procedure entry as done for procedures with a single *alloc* instruction, register stack unwinding is unharmed.

5. Experimental Evaluation

5.1. Methodology

Using the Intel[®] Itanium[®] production compiler and an Itanium[®]-2 system, data was collected on twelve C/C++ programs from the SPEC CINT2000 benchmark suite. The Itanium[®]-2 processor [10] contains two integer units, two load/ALU units, two store/ALU units, and three branch units. Integer multiplies and divides can be executed on the processor's two floating-point units, adding additional integer throughput for some programs. The processor has a three-level on-chip cache hierarchy. The SPEC CINT2000 benchmarks were aggressively compiled using the switches that are used for a "base" SPEC build including: classical

optimizations, scheduling with control and data speculation, predication, loop optimizations and data prefetching, profile-guided optimizations, and interprocedural optimizations. The multiple allocation optimization was instrumented to collect data on the average amount of stack frame size reduction, the number of registers allocated, and the number of registers live across the call.

5.2. Results and Analysis

Four categories of results are presented. First, an analysis of the call site density relative to the *alloc* instruction density. Next, the number of allocated registers in a stack frame are compared against the number of live registers in the stack frame. Then, the run time performance of the new optimization relative to the default Intel[®] Itanium[®] compiler. Finally, the latency due to RSE stalls is examined.

5.2.1. Procedure Call Site Characteristics

Table 1 lists the number of non-leaf functions in each benchmark (#cfunc), and the number of these that can use multiple *alloc* instructions (#mfunc). A leaf function is a function that does not contain any call sites. These numbers indicate that the algorithm has the potential to insert multiple *alloc* instructions in 13%-43% of all non-leaf functions. Thus, if an aggressive *alloc* placement policy is used, the static distribution of these instructions will cover a significant number of functions. The run time behavior of each benchmark could benefit significantly from these *alloc* instructions, provided that call blocks that use the optimization are frequently executed and trigger RSE spills. For the case of 252.eon, the multiple *alloc* technique can reduce the register stack frame in 85% of all non-leaf functions.

Benchmark	#cfunc	#mfunc	#mfunc / #cfunc
164.gzip	75	18	24%
175.vpr	209	67	32%
176.gcc	112	43	38%
181.mcf	13	2	15%
186.crafty	78	31	40%
197.parser	281	95	34%
252.eon	81	69	85%
253.perlbnk	1108	214	19%
254.gap	742	106	14%
255.vortex	726	95	13%
256.bzip2	58	13	22%
300.twolf	134	57	43%

Table 1. Non-leaf functions with multiple *alloc* instructions.

In order to gain further intuition into the potential performance improvements that this optimization may offer, the

number of call sites in each benchmark should also be examined. Figure 7 displays the average number of call sites for the functions in each benchmark. With benchmarks such as 186.crafty and 176.gcc averaging more than 25 call sites per function, there appears to be a significant amount of RSE activity, and a strong likelihood that RSE spilling and filling will occur.

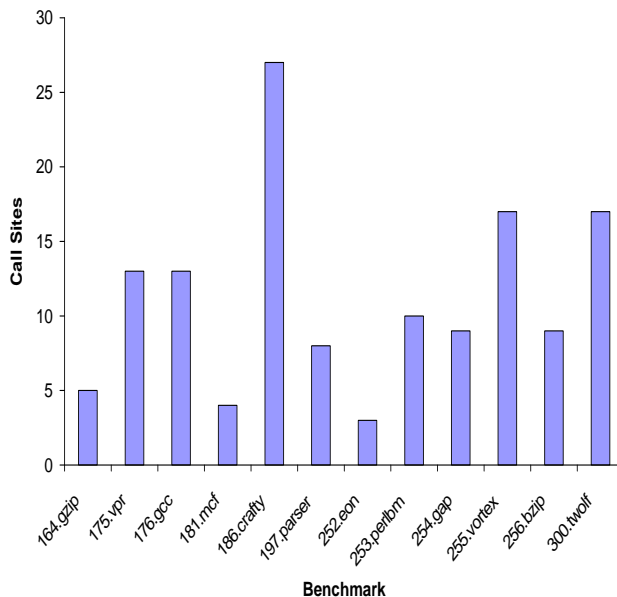


Figure 7. Number of call sites per function.

5.2.2. Live Registers vs. Allocated Registers

After *alloc* instructions have been issued, the number of remaining live registers in the stack frame can be significant. Figure 8, displays the difference between the number of stacked registers allocated after the multiple *alloc* to the number of registers live across the call site. Across the benchmarks, the frame size could be reduced by an additional 5 registers. Thus, for most cases, it appears that when the *alloc* instruction is issued, it does a relatively good job of reducing the stack frame size. However, the data indicate that a multiple *alloc* aware register graph-coloring algorithm [5], could further decrease the stack frame size by at least 5%.

Figure 9 indicates that call sites that use the *alloc* instruction save a significant number of registers. The figure shows the average distance between the initial stack frame size and the maximum frame size allocated across call sites. This worst-case distance is relatively high, 10-15 registers for all benchmarks on the average. This implies that when the *alloc* instruction is inserted, that it can reduce the the stack frame size significantly.

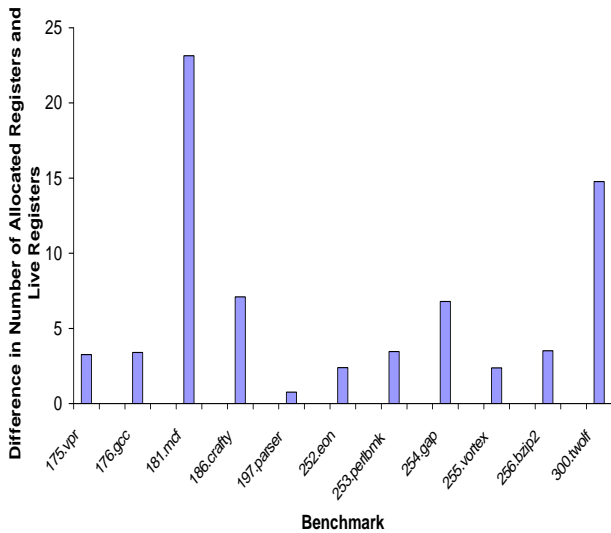


Figure 8. Ratio of live registers to allocated registers.

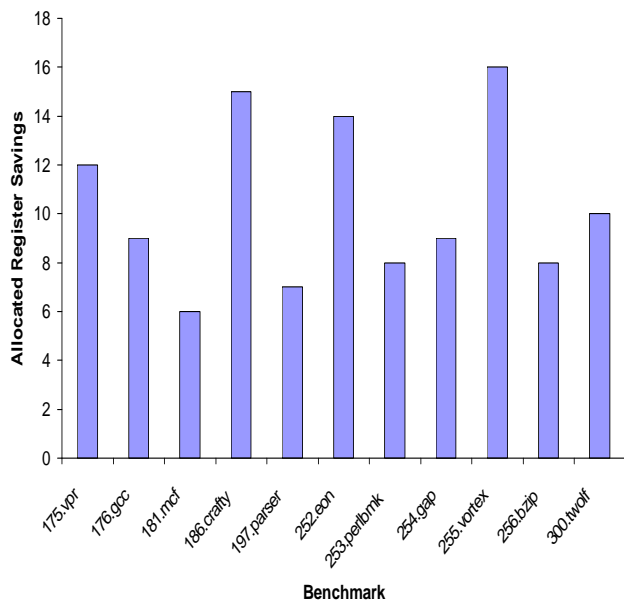


Figure 9. Average distance between max alloc frame and original frame.

Figure 10 shows the proportion of issued *alloc* instructions relative to the number of call sites for each benchmark. In most cases, the algorithm issues *alloc* instructions for roughly 10% of all call sites. This could lead to significant run-time performance gains provided that the call sites in question are frequently executed. The data also

suggests that there is room for improvement for the *alloc* placement algorithm. Benchmarks such as 186.crafty and 252.eon both have functions that allocate the maximum allowable stack frame size (96), so they are likely to trigger RSE spills. Increasing the number of call sites where *alloc* instructions are issued would increase the chances of reducing RSE spills and fills.

If the register allocation phase could be tuned to be multiple *alloc* aware, then the *alloc* to *call* ratio would likely increase. This would result from local registers being placed closer to the incoming registers, giving the algorithm a greater register set to choose from. The most significant issue in regards to the live register set is that in many cases, the *alloc* instruction is not issued simply because live local registers are located at the top of the stack frame.

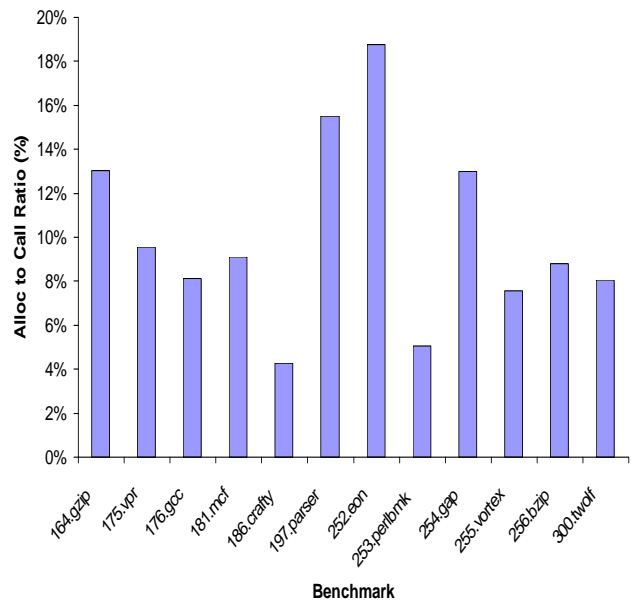


Figure 10. Alloc to subroutine call ratio.

5.2.3. Run-time Performance

The run-time performance of this optimization was compared against the Intel[®] Itanium[®] compiler with the base set of optimizations enabled. The data in Figure 11 indicates that the performance of the compiler with this optimization is roughly identical to the performance without. These results are encouraging for an initial implementation of a compiler optimization because they indicate that the optimization has not adversely affected the compiler. Similarly, the accompanying data suggests that there are more opportunities to improve upon the allocation algorithm.

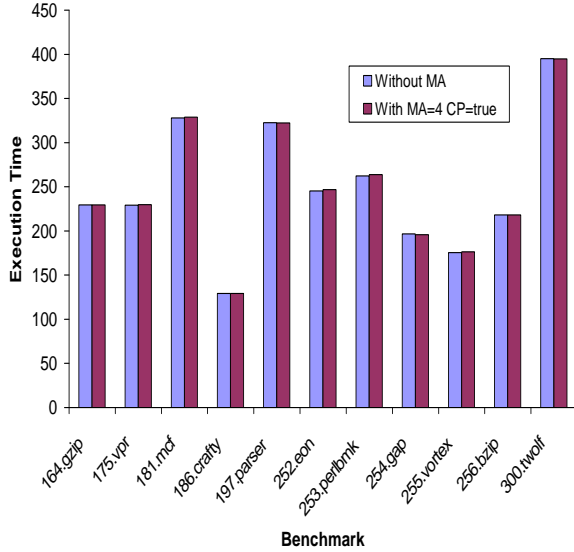


Figure 11. Execution time with and without multiple alloc optimization.

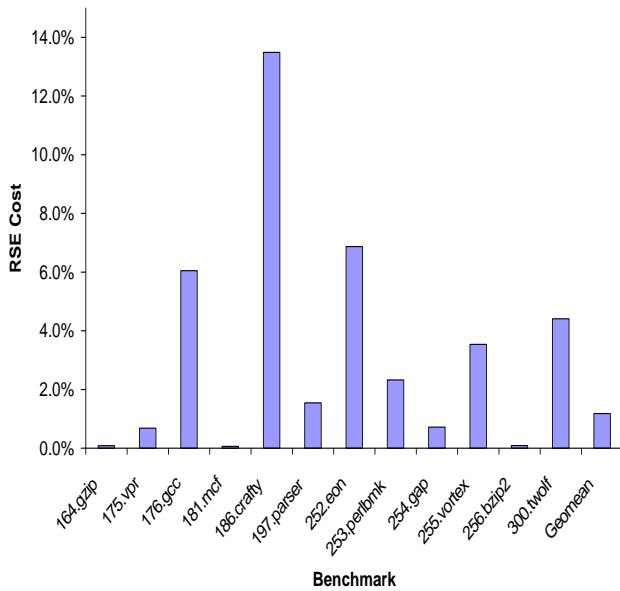


Figure 12. Latency incurred by RSE.

5.2.4. RSE Latency

Figure 12 displays the percentage of stall time due to RSE traffic. For many benchmarks RSE spills and fills are not likely to occur. However, 176.gcc, 186.crafty, and 252.eon all spend more than 6% of their execution time stalled for RSE transfers. These benchmarks can benefit significantly

from improved RSE performance, and provide the motivation for implementing a more aggressive multiple alloc optimization.

6. Summary

This implementation of the multiple alloc optimization demonstrates that it is possible to use the compiler to improve RSE performance. The data indicate that there are many call sites where the number of allocated registers are significantly greater than the number of live registers. The multiple alloc optimization can reduce the average stack frame size per procedure significantly. Ultimately, this optimization technique is an alternative to increasing the size of the physical register file of future Itanium[®] implementations. Results indicate that using this new optimization technique can save an average of 8-10 registers per stack frame from being spilled to memory by the Itanium[®] RSE. It is expected that applications with substantial subroutine recursion and subroutine traversals will extract even greater benefit from the new optimization technique.

Future work will be directed at making the algorithm more creative in locating positions in the control flow graph where alloc instructions could be inserted. Currently, the algorithm is bound to *call* instructions, and does not weigh the overhead required in issuing the *alloc* instruction relative to the performance improvement that may or may not result. However, the algorithm is conservative in that it does not attempt to allocate a register stack frame at every basic block, which could increase the code size dramatically. A profile-guided approach to leveraging the execution weights of blocks with subroutine calls could also yield improvements to the new optimization [8]. Finally, the graph-coloring based register allocator [5][3] will be made multiple alloc aware.

Acknowledgments

Steve Skedzielewski provided the RSE run time performance data. Mark Davis and the anonymous reviewers helped improve the quality of this paper significantly with their critical comments. Alban Douillet had implemented a prototype of the multiple alloc algorithm during his internship in summer 2000.

References

- [1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predication and speculative execution in

- the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–237, June 1998.
- [3] J. Bharadwaj, W. Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce. The intel ia-64 compiler code generator. *IEE Micro*, 20(5):44–52, September, October 2000.
- [4] I. Bratt, A. Settle, and D. A. Connors. Predicate-based transformations to eliminate control and data-irrelevant cache misses. In *Proceedings of the First Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techniques*, pages 11–22, December 2001.
- [5] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symp. on Compiler Construction*, pages 98–105, June 1982.
- [6] A. Douillet, J. N. Amaral, and G. R. Gao. Fine-grain stacked register allocation for the itanium architecture. In *15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2002.
- [7] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 158–168, December 1995.
- [8] G. F. Hoflehner and J. E. Pierce. Method and apparatus for inserting more than one allocation instruction within a routine. In *United States Patent Disclosure*, June 2002.
- [9] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual*. Santa Clara, CA, 2000.
- [10] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual*. Santa Clara, CA, 2002.
- [11] D. Keppel. Register windows and user-space threads on the SPARC. Technical Report TR-91-08-01, 1991.
- [12] T. Kiyohara, S. M. W. Chen, R. Bringmann, R. Hank, S. Anik, and W. Hwu. Register connection: A new approach to adding registers into instruction set architectures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 247–256, May 1993.
- [13] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C. Lim, J. Ng, and D. Sehr. An advanced optimizer for the ia-64 architecture. *IEEE Micro*, 20(6):60–68, November 2000.
- [14] M. Martin, A. Roth, and C. Fischer. Exploiting dead value information. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 125–135, December 1997.
- [15] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, San Francisco, California, 1997.
- [16] M. Postiff, D. Greene, S. Raasch, and T. N. Mudge. Integrating superscalar processor components to implement register caching. In *International Conference on Supercomputing*, pages 348–357, 2001.
- [17] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.
- [18] D. L. Weaver and T. Germond. *The SPARC Architecture Manual*. SPARC International, Inc., Menlo Park, CA, 1994.
- [19] R. D. Weldon, S. S. Chang, H. Wang, G. Hoflehner, P. H. Wang, D. Lavery, and J. P. Shen. Quantitative evaluation of the register stack engine and optimizations for future itanium processors. In *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, Santa Clara, CA 95052, July 2002.