

Performance Advantage of the Register Stack in Intel® Itanium™ Processors

Ryan Rakvic, Ed Grochowski, Bryan Black, Murali Annavaram, Trung Diep, and John P. Shen

*Microprocessor Research, Intel Labs (MRL)
2200 Mission College Blvd
Santa Clara, CA 95052
ryan.n.rakvic@intel.com*

Abstract

The Intel® Itanium™ architecture provides a virtual register stack of unlimited size for use by software. New virtual registers are allocated on a procedure call and deallocated on return. Itanium processors implement the register stack by means of a large physical register file, a mapping from virtual to physical registers, and a Register Stack Engine (RSE) that saves and restores the contents of the physical registers to memory without explicit program intervention. The combination of these features significantly reduces the number of loads and stores required to save registers across procedure calls compared to a conventional architecture. In this paper, we show that the Itanium register stack reduces load and store traffic to the stack by at least a factor of three across select SpecInt2000 and Oracle database benchmarks. Furthermore, we examine the effects of the register stack on data cache miss rates and program execution time. When compared to a conventional architecture, the Itanium architecture on average achieves 7%-8.3% and 10.2%-12% performance advantage on in-order and out-of-order processor models, respectively, as a result of the register stack. Finally we analyze the vitality of stack loads and show that in general few stack loads are vital in an in-order model. However, a larger percentage of stack loads become vital in the out-of-order model leading to a greater performance benefit from the register stack.

1. Introduction

The Intel® Itanium™ architecture contains a number of innovative features designed to speed up program execution. These features include:

1. Explicit parallelism: a mechanism whereby the compiler identifies groups of independent instructions for parallel execution by hardware
2. Control speculation: a mechanism to move loads and their dependents ahead of conditional branches
3. Data speculation: a mechanism to move loads and their dependents ahead of potentially conflicting stores
4. Predication: conditional execution of instructions, thereby avoiding mispredicted branch penalties

5. Register stack: a large physical register file organized as a stack with hardware to automatically save and restore registers to memory

An overview of the Itanium architecture may be found in [1] and [2]. A detailed description may be found in [3]. As the Itanium architecture is very new, we are just beginning to understand the performance implications of each of these features. An in-depth analysis of the effectiveness of predication was presented in [4] showing very limited performance advantage. In this paper, we focus on the register stack. We describe the features of the register stack, while highlighting both the positives and negatives. We show that the Itanium register stack provides an overall performance advantage when compared to a similar processor without one. This is not a paper proposing a new idea for a future design, but an evaluation of a known technique that has actually been implemented in a real machine for a new architecture. The goal of this paper is to assess how well the Itanium register stack performs and to explain why. To our knowledge, no previous technical paper has provided a detailed performance evaluation of a register stack architecture.

The outline of this paper is as follows. Section 2 presents an overview of the Itanium register stack. Section 3 presents the simulation methodology used in this paper. We then compare the Itanium register stack to a conventional register file, noting the register stack's effects on dynamic instruction counts in Section 4, cache miss rates in Section 5, and execution time in Section 6. Finally, we analyze the vitality of stack loads in Section 7.

2. Itanium Register Stack

Overlapping register windows have long been used in RISC architectures to store the local variables required by multiple procedures in a large on-chip register file [5,6,7]. Fixed size, two size, and variable size register windows have been studied in [8], and hardware register windows have been compared against software register allocation in [9]. Prior to the Itanium Processor Family (IPF), the only major commercial microprocessors that employ register windows were the SPARC microprocessors. This section presents an overview of the Itanium register stack.

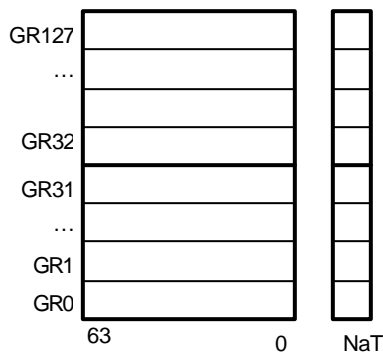


Figure 1 Itanium Integer Register File.

The Itanium architecture specifies 128 architected 64-bit integer registers as shown in Figure 1. Each register stores 64-bits of integer data along with a single NaT (*Not a Thing*) bit that indicates a deferred exception has occurred during control speculation. The 128 architected integer registers are divided into two groups: 32 static registers GR[0:31] and 96 stacked registers GR[32:127]. The static registers are available to all procedures while the stacked registers are allocated on demand to each procedure. GR0 is always read as zero. A subset of the stacked registers may be used as rotating registers during software pipelined loops; the rotating subset is considered part of a procedure’s local variables.

Register allocation works as follows. Each procedure is responsible for allocating a stack frame consisting of input parameters, local variables, and output parameters. This is accomplished via the *alloc* instruction. Stack frames overlap so that the registers holding the output parameters of the caller become the input parameters of the callee. The overlap is illustrated in Figure 2.

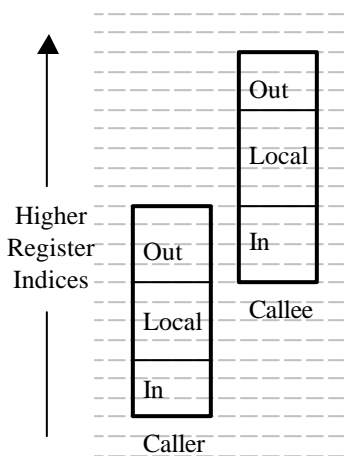


Figure 2 Overlapping Stack Frames.

Due to the overlap, each procedure needs to specify on an *alloc*:

1. The number of registers needed for input parameters and local variables
2. The total number of registers needed by the procedure’s stack frame, equal to the sum of input, local, and output variables

Each procedure’s stack frame may contain any number of registers up to a maximum of 96. A new stack frame is created by a procedure call and the associated *alloc*. The frame is automatically deallocated on the procedure return. Unlike stacks in other computer architectures, the register stack in the Itanium architecture grows *upwards* towards higher register indices and higher memory locations.

While the Itanium architecture specifies 96 architected stacked registers, the actual number of physical stacked registers in an implementation may be larger than 96. The physical register stack must contain at least 96 registers in order to support the ability of a procedure to allocate up to a maximum of 96 registers in its stack frame. The physical register stack in the current Itanium processors contains the minimum 96 stacked registers. Future implementations may have a larger physical register stack.

What happens when the total number of registers allocated in all stack frames exceeds the size of the physical register stack? The Itanium architecture defines a hardware Register Stack Engine (RSE), which transfers the contents of the physical registers to and from memory in order to create the illusion of an infinite-size register stack. The area of memory reserved for this purpose is called the *backing store*; the act of storing registers to the backing store is called a *spill*, and the act of loading registers from the backing store is called a *fill*. The relationship between the register stack and the backing store is illustrated in Figure 3.

The physical register stack is partitioned into three regions. The invalid region contains not-yet-allocated registers. The active region is used by the current procedure to hold input parameters, local variables, and output parameters. Below the active region is the dirty region. These registers hold the stack frames belonging to the current procedure’s *callers* that have not been written to memory. Once spilled to memory, a register becomes part of the invalid region for subsequent use by the current procedure’s *callees*.

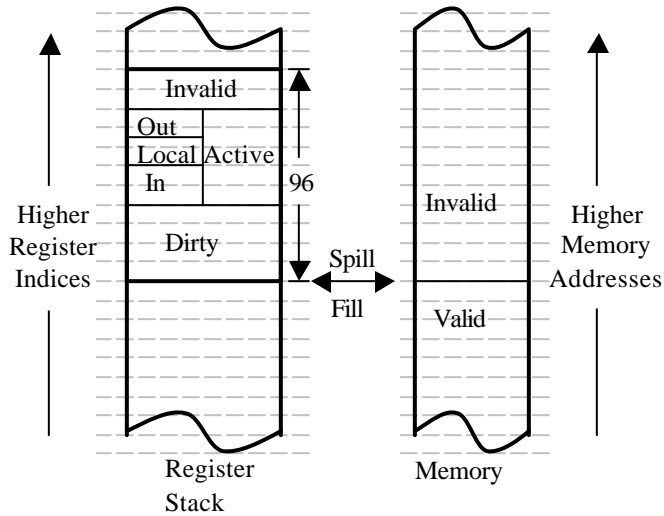


Figure 3 Relationship between Register Stack and the Backing Store in memory.

Even though the physical register stack has a fixed size, the RSE’s job is to maintain the illusion of an infinite-size register stack by spilling and filling registers to memory without explicit control by the application program. The physical register stack is maintained as a circular buffer by the RSE and functions as a “window” into the infinite-sized virtual register stack. One RSE algorithm is as follows:

1. When a procedure allocates a new stack frame, if the top of the frame (active region) extends above the top of the physical register stack window, then the window is moved up by spilling some dirty registers to the backing store. These dirty registers belong to the current procedure’s *callers*.
2. After a procedure returns and its stack frame is discarded, if the bottom of the caller’s frame (now the active region) extends below the bottom of the physical register stack window, then the window is moved down by filling registers from the backing store. These registers belong to the current procedure.
3. Procedure calls, *allocs*, and returns within the physical register stack window do not need spills/fills.

Note that RSE spills and fills to the backing store are subject to address translation and may incur memory exceptions. The Itanium architecture requires that exceptions resulting from RSE activities be precisely reported.

Figure 4 illustrates the actions of the RSE during the execution of a 20K instruction excerpt from *gcc*. As the program’s call depth changes, the RSE moves the location

of the register window so that the current frame is always mapped within the window (containing 96 physical registers in this simulation). Moving the window up generates spills while moving the window down generates fills. Not all changes in call depth require movement of the register window. The ability to keep the register window unchanged across multiple procedure calls leads to a reduction in load/store traffic. The rest of this paper analyzes this effect, and shows the overall impact of the register stack.

3. Simulation Methodology

This section describes the simulation methodology used in this paper. For workloads, we select from the SPEC CPU2000 integer suite seven benchmarks that exhibit reasonable amounts of procedure call activity: *crafty*, *gap*, *gcc*, *gzip*, *parser*, *perlbnk*, and *vortex*. In addition, we include the *Oracle Database Benchmark* [10] as an example of a large transaction processing workload.

All benchmarks are compiled with the Intel Electron compiler for the Itanium Processor Family. We run 250M instruction samples of each benchmark using two Itanium instruction set simulators: SoftSDV [22] for the CPU2000 benchmarks and Simics [23] for the *Oracle* benchmark. The CPU2000 benchmarks primarily consist of applications code, whereas *Oracle* has significant contributions from both the application and operating system that are included in our trace. The results of the instruction set simulators are fed to a cycle-accurate microarchitecture performance simulator that implements the machine configuration summarized in Table 1.

Pipeline Depth	8 cycle branch misprediction penalty
Fetch Width	2 bundles
Branch Predictor	512 entry BTB 4096 entry PHT 8 bit local history
Register Files	128 Integer Registers 128 FP Registers 64 Predicate Registers
Execution Resources	6 integer ALUs 2 load 2 store
Cache Structure	L1I: 16K 4-way, 1 cycle latency L1D: 16K 4-way, 1 cycle latency L2: 256K 4-way, 6 cycle latency L3: 3072K 12-way, 12 cycle latency All caches have 64 byte lines
Main Memory	100 cycle latency

Table 1 Research Itanium Machine Model Parameters.

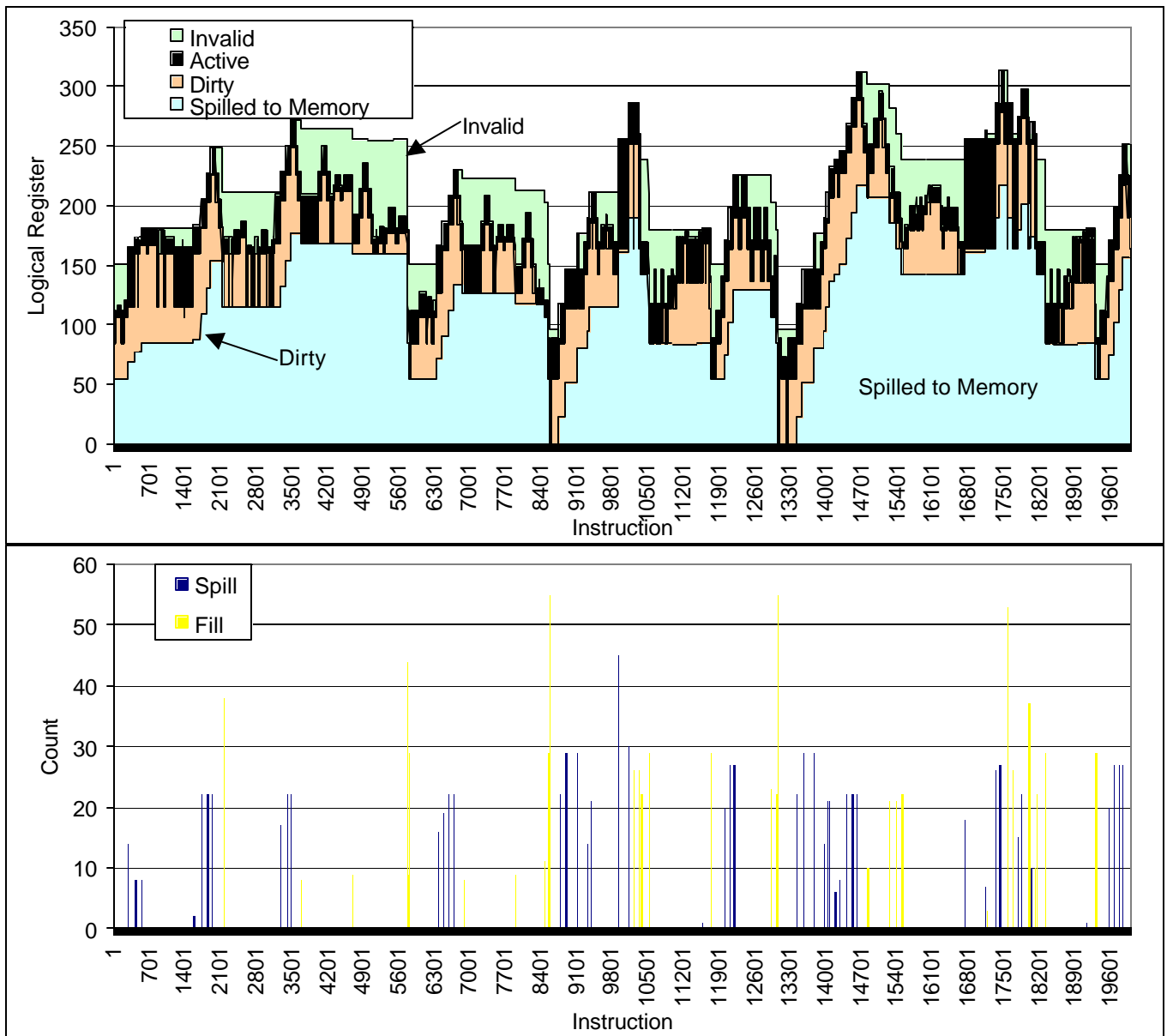


Figure 4 Snapshot of the Itanium Register Stack in Operation.

While the activities of the register stack may be measured using the performance-monitoring features of the Itanium processor, we use simulation to enable studies of arbitrary register stack configurations. We consider both in-order and out-of-order implementations of the Itanium architecture. Although current IPF implementations are in-order, out-of-order IPF processors may be implemented in the future using the techniques described in [11].

To compare the performance impact of the register stack, we simulate two distinct machine models:

1. One with the register stack (IPF)
2. One without the register stack (conventional).

For the conventional model, we modified our simulator's RSE to perform saves and restores of all registers used for local variables in each procedure. The resulting memory traffic is similar to what a conventional architecture with 128 architected registers and callee-save conventions would experience. In the next section we begin by showing the increase in memory traffic in terms of extra loads/stores for the conventional machine relative to the IPF machine with the register stack.

	crafty	gap	gcc	gzip	parser	perlbnk	vortex	oracle	total
Loads	13.13%	16.07%	14.72%	15.65%	17.78%	19.96%	15.90%	17.00%	16.21%
Stores	2.98%	6.93%	5.24%	8.09%	3.59%	7.82%	8.28%	8.12%	6.31%
Calls	0.27%	0.64%	1.14%	0.28%	0.68%	1.72%	1.12%	0.84%	0.83%
IPF-96 spills+fills	10.42%	0.30%	6.29%	0.00%	1.29%	0.29%	2.75%	7.74%	3.92%
IPF-128 spills+fills	5.11%	0.06%	1.52%	0.00%	0.67%	0.00%	1.22%	4.70%	1.80%
IPF-192 spills+fills	1.82%	0.02%	0.25%	0.00%	0.18%	0.00%	0.33%	2.13%	0.64%
Conventional saves+restores	16.49%	18.38%	22.01%	8.49%	11.62%	21.46%	8.30%	14.63%	15.26%

Table 2 IPF and Conventional Dynamic Instruction Counts.

4. Dynamic Instruction Counts

Table 2 presents dynamic instruction counts for the CPU2000 and *Oracle* benchmarks. The instruction counts are expressed as a percentage of the total non-NOP retired instructions for the *IPF-96* configuration. On IPF processors, NOPs account for 28% of all retired instructions. The NOPs are the result of the instruction templates, which allow only certain instruction types in each of the three instruction slots, and branch targets that are aligned to boundaries of three-instruction bundles. Total non-NOP retired instructions for the *IPF-96* configuration are used as the denominator for all configurations. The number of RSE spills is equal to the number of RSE fills; the results in the table show the total of the two. Similarly, the number of register saves is equal to the number of register restores, and is half of the total shown. We refer to IPF stack stores and loads as spills and fills respectively, and to conventional architecture stack stores and loads as saves and restores respectively.

As shown in Table 2, loads account for 16.2% of all retired instructions; stores account for 6.3%, and procedure calls account for 0.8%. The percentage of loads and stores (22%) in the Itanium architecture is significantly lower than other architectures due to the large physical register file and the register stack. The number of RSE spills and fills with 96 physical stacked registers is shown in the *IPF-96* row. RSE spills account for only 2.0% of all instructions, and RSE fills account for another 2.0%. Although the contribution of the RSE to the total instruction count is small, there is considerable variation among the benchmarks. *Crafty*, *gcc*, and *Oracle* contain high levels of RSE activity, while *gzip* contains essentially no RSE activity.

Overall, having 96 physical stacked registers is quite effective in minimizing spill/fill traffic. As shown in the *IPF-128* row, increasing the number of physical stacked registers from 96 to 128 can reduce the number of spills and fills by a factor of two to just 1.8%. Further increasing the physical register stack size to 192 can reduce the spill/fill traffic to 0.6%. This indicates the true effectiveness of the register stack.

As stated in Section 3, we also simulate the number of stack register saves and restores that would be required in a conventional architecture. This is accomplished by modifying our simulator's RSE to save and restore all local registers in each procedure's stack frame as indicated by the *alloc* instruction. We assume that only local variables need be saved and restored. Registers containing input and output parameters, as well as the 32 static registers, are not preserved by the RSE. The resulting memory traffic approximates the behavior of a conventional architecture with 128 architected registers and callee-save conventions.

We observe that in a conventional architecture, saves and restores account for 15.3% of all instructions. Comparing the *IPF-96* against the *Conventional* configuration, we see that the Itanium register stack is successful in reducing the memory traffic required to save and restore registers across procedure calls by a factor of 3.9. With *IPF-128* this factor becomes 8.5. Comparing the total number of loads and stores performed by the *IPF-96* and *Conventional* machines, we note that *IPF-96* would have experienced a 31% increase in load traffic and 68% increase in store traffic in the absence of the register stack. In subsequent sections, we'll evaluate the Itanium architecture using the *IPF-96* and *Conventional* configurations, referring to these configurations simply as *IPF* and *Conventional* respectively.

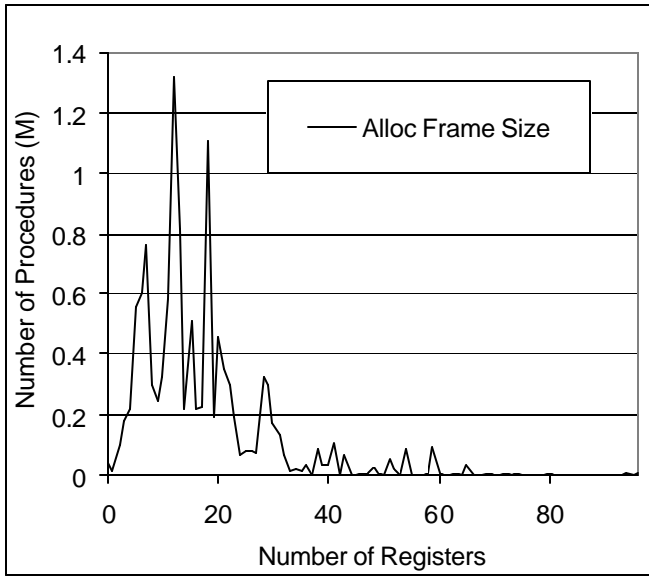


Figure 5 Number of Dynamic Procedures vs Number of Stacked Registers.

Unlike the fixed-size register windows in the SPARC architecture, the Itanium register stack allows the flexible allocation of arbitrary number of registers for each stack frame. Figure 5 presents the distribution of the total number of registers allocated in each procedure's stack frame. This data is summed across the CPU2000 and *Oracle* benchmarks. The most frequently occurring stack frame allocates 12 registers (in addition to the 32 static registers not included in the graph). Furthermore, 90% of all procedures allocate fewer than 30 registers. This data indicates that the 96 physical stacked registers in the present IPF implementations can be expected to hold the stack frames for multiple procedures and keep the number of spills and fills to a minimum.

One limitation of the present compiler is that it allocates once (at the beginning of each procedure) the maximum number of registers required by all possible control flow paths through that procedure. Depending on the actual control flow path, not all allocated registers may be used by the procedure. Table 3 presents the ratio of the registers used by the actual control flow versus the number allocated at the beginning. On average, 76% of all allocated registers are used by the actual control flow. In both the Itanium and conventional architectures, it is possible for the compiler to implement either one allocation (or block of saves and restores) per procedure, or multiple finer-granularity allocations (or blocks of saves and restores) at different points within the procedure. For our comparison, we assume one allocation per procedure for both the *IPF* and *Conventional* configurations.

crafty	gap	gcc	gzip	parser	vortex	average
78.7%	59.8%	69.1%	80.1%	90.2%	84.3%	75.8%

Table 3 Ratio of Demand to Allocated Register Usage.

As previously stated, the Itanium architecture specifies a minimum of 96 physical stacked registers. Figure 6 shows the effects of varying the number of physical stacked registers on the number of RSE spills. As shown in Figure 6, the number of spills (and corresponding number of fills) can be further reduced, by increasing the number of physical stacked registers beyond 96. The number of spills increases dramatically with fewer than 96 physical registers.

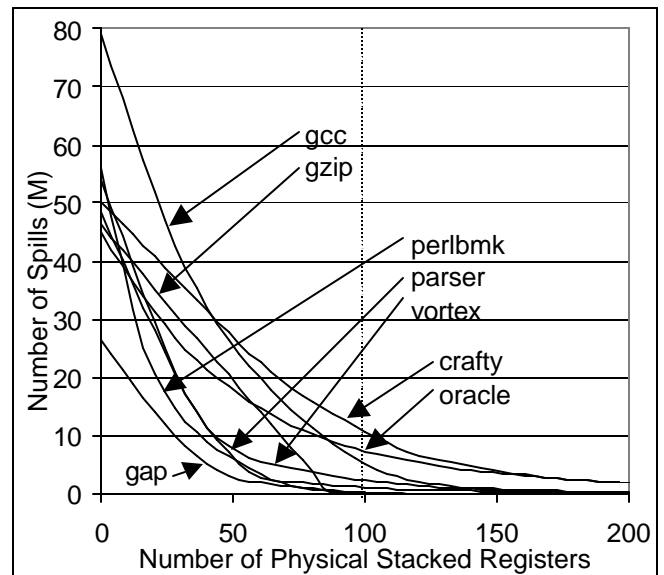


Figure 6 Varying the Number of Physical Stacked Registers.

5. Cache Impact of the Register Stack

This section examines the effects of the Itanium register stack on data cache miss rates and the number of misses. It is well known that the performance of a microprocessor is highly dependent on the performance of its cache memory. Most try to design a Data Level 1 Cache (L1D) that maintains a low miss rate with very low latency [12]. Current approaches include using streaming buffers,

victim caches [13], alternative cache indexing schemes [14], etc. [15].

Section 4 shows that the register stack eliminates many stack loads and stores. We first analyze cache impact of the register stack by detailing the miss rate of the loads that are accessing the stack (we refer to IPF stack loads as *fills* and conventional architecture stack loads as *restores*). These stack loads share the L1D cache with non-stack (heap) loads, but for now we separate their miss rates. Figure 7 presents the L1D miss rates of only these stack loads. (Recall that the L1D cache is fixed at 16KB.) As can be seen, the miss rate for the stack loads is very low for both configurations. On average, the miss rate for stack loads is below 3%. For most of the benchmarks, the conventional configuration has a lower miss rate than the IPF configuration. The conventional configuration has many more stack references, and this leads to better locality and a lower miss rate.

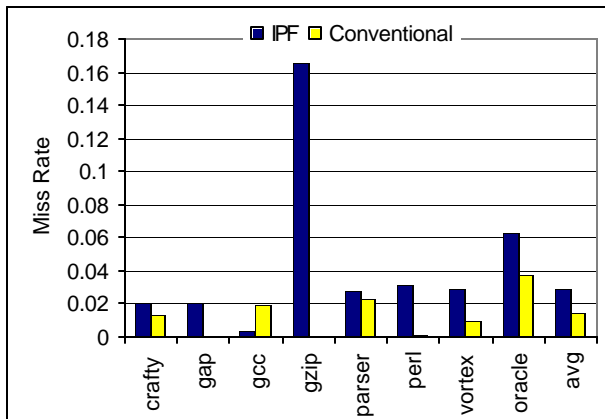


Figure 7 L1D Cache Miss Rate of Stack Loads.

Although the miss rate is lower for the conventional configuration, the total number of misses from stack loads is greater for most of the benchmarks. Figure 8 illustrates the overall number of misses generated by stack loads normalized to the IPF configuration. For example, for *perl*, the conventional configuration has over 4.3 times the number of stack load misses than the IPF configuration. On average, the conventional configuration has approximately 90% more overall stack load misses than the IPF configuration. However, as we saw in Figure 7, the stack loads have a very low miss rate and hence low overall misses. Therefore, the register stack’s cache impact does not significantly impact overall

performance.

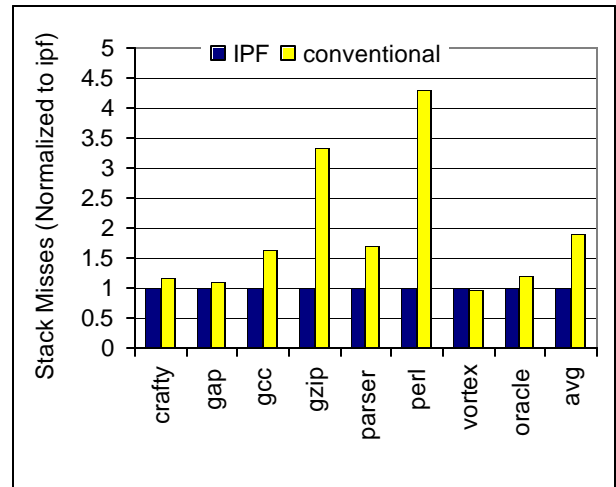


Figure 8 Overall Stack Load Misses.

We now analyze the impact the register stack has on the overall number of load misses (stack & heap). Figure 9 presents the overall number of load (stack & heap) misses (normalized to IPF) for all the benchmarks. The conventional configuration introduces more stack bads accessing more data space, and therefore increases the total number of misses. Although the stack loads’ miss rates are low, they still add more overall misses since a majority of those loads do not exist in the IPF configuration. Also, the extra stack loads add extra data contention to the L1D. For example, for the benchmarks *crafty* and *Oracle*, they experience a similar number of stack misses (Figure 8), but the total number of L1D misses increases significantly. However, it can be seen that the total number of misses does increase, but not significantly for most of the benchmarks because the

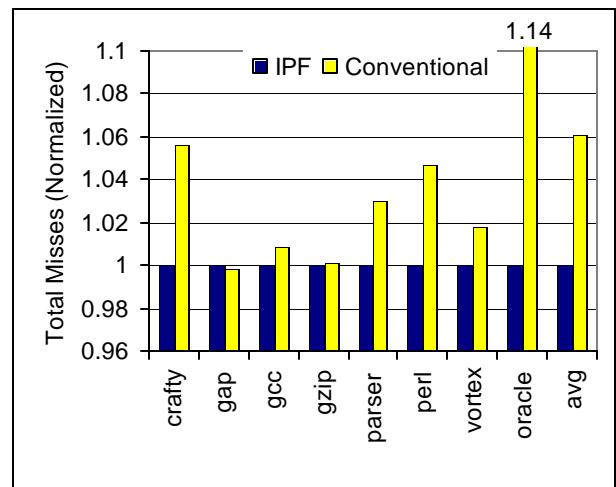


Figure 9 L1D Total Load Misses.

working set of the stack is miniscule relative to the 16KB L1D. Furthermore, the total number of misses of the L1D is small to begin with, and the overall cache impact of the register stack is also small.

6. Performance Advantage of the Register Stack

Up to this point, we have considered only secondary effects of the register stack in IPF processors. Now we consider overall performance impact of the register stack in terms of speedup. Once again we compare the Itanium-96 (IPF) model with a similar machine with no register stack (conventional).

Figure 10 presents the speedup, in terms of total execution cycles, of the IPF configuration compared to the conventional configuration. Both configurations are simulated on an in-order model, and assumed to have the parameters of Table 1. The performance benefit of the register stack ranges from 1.7% - 11.9% averaging 7%.

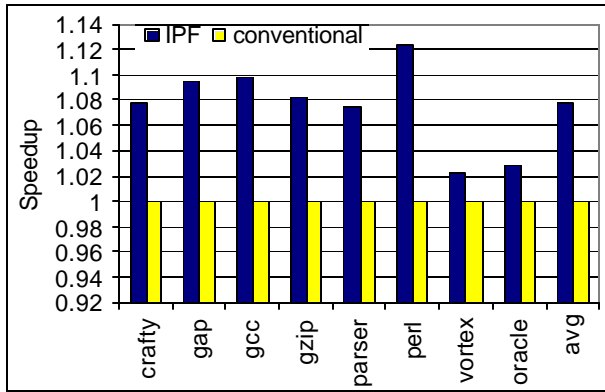


Figure 10 IPF-96 Performance Advantage (In-Order).

The register stack in the Itanium architecture reduces the total number of stack references, and hence provides a performance advantage compared to the same machine without a register stack. The performance advantage of the register stack can be attributed to the reduction in overall instructions. Table 2 presents the overall difference in instruction counts. In these in-order machines, there is strong correlation between instruction count and overall execution time since the in-order machine does not overlap the execution of stack loads and stores with a program's other instructions. However for the *Oracle* benchmark, the performance impact does not directly correlate with the increase in spill/fill count. *Oracle* spends a lot of time waiting for memory (and exhibits a lower IPC), and the increase in overall instructions has less of an effect on performance.

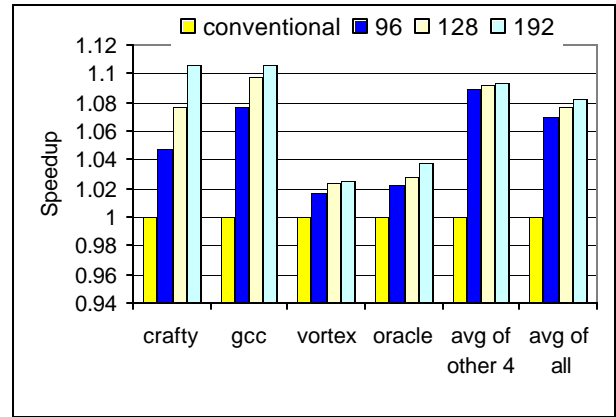


Figure 11 Register Stack Size Impact on Performance (In-Order).

The performance numbers presented thus far assume a physical register stack size of 96. This number can be increased in actual implementations. As Table 2 illustrates, a few of the benchmarks could potentially benefit from an increase in physical register stack size. Figure 11 provides the speedup achieved for various (96, 128, 192) stack sizes and compares them to the conventional machine. For the benchmarks *crafty*, *gcc*, *vortex*, and *Oracle*, an increase in the stack size from 96 to 128 increases performance greatly. Furthermore, for *crafty* and *Oracle*, an increase in the stack size to 192 still significantly increases performance. For instance, *Oracle's* performance gain over the conventional machine increases from 2.2% (96 stacked registers) to 3.8% (192 stacked registers). We can conclude that *Oracle* is using a lot of registers in a short amount of time. For the other 4 benchmarks (*gap*, *gzip*, *parser*, *perl*), a register stack size of 96 is more than enough to reduce spills/fills and hence performance is not increased significantly with an increase in register stack size. The overall average performance benefit of the register stack ranges from 7% (96 stacked registers) to 8.3% (192 stacked registers) for the in-order machine.

The register stack is an architectural feature that can be applied to an in-order or an out-of-order implementation. So far we have presented numbers for an in-order model. Now we present the performance advantage of the register stack in the context of an out-of-order model. Once again, we show speedup for the two configurations (IPF and conventional). Figure 12 presents the speedup for the benchmarks. The performance advantage of the register stack is slightly higher in the out-of-order model compared to the in-order model. In the out-of-order model, the register stack averages a 10.2% increase in performance for all benchmarks for *IPF-96*. For *IPF-192* (not in figure), the performance gain is 12%.

One would expect the out-of-order model to mask some of the performance advantage of the register stack because independent instructions can execute and use the available resources while the stack loads and stores execute concurrently. However, as shown in Figure 12, the out-of-order model does not reduce the performance advantage of the register stack. In fact, the opposite is true: the register stack provides a greater performance benefit in the out-of-order machine. As we will see in Section 7, the extra stack loads in the conventional model are vital instructions [17], and dependent instructions are forced to wait for these stack loads to execute.

We see in Section 5 that the cache effects of the register stack are minimal. The extra stack loads do not increase the overall number of misses experienced by the L1D significantly. Additionally, we find that the latency of the L1D has minimal impact on the extra stack loads. However, the bandwidth for executing stack loads and stores does have an impact on the performance advantage of the register stack.

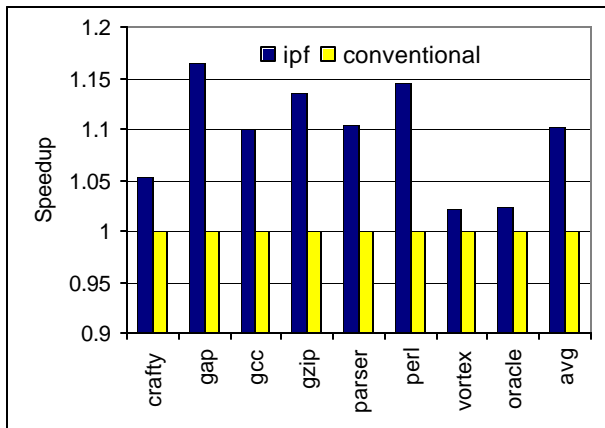


Figure 12 IPF-96 Performance Advantage (Out-Of-Order).

[16] cleverly provided high-bandwidth and low-latency caches by decoupling accesses of stack variables from heap variables. The decoupled caches provide more bandwidth when stack variable accesses are intermingled with non-stack accesses. The design requires identification of local accesses either statically or dynamically. Our two configurations use the L1D for both stack and heap data. So far we fixed the number of ports for the L1D at four (2 Read & 2 Write). This allows for 2 loads and 2 stores to be executed in parallel. Now we simulate the same two configurations (IPF and conventional) with the number of ports for the L1D set at two (1 Read & 1 Write). This allows only 1 load and 1 store to be executed in parallel. Figure 13 presents the speedup of the register stack under these constraints. Comparing Figure 10 and Figure 13 it can be seen that the performance advantage of the register stack increases

when the bandwidth of the L1D is reduced. For the in-order machine, the overall benefit of the register stack increases from 7% to 9% on average. When peak bandwidth is limited, the conventional configuration suffers a bigger performance penalty due to the greater number of stack loads.

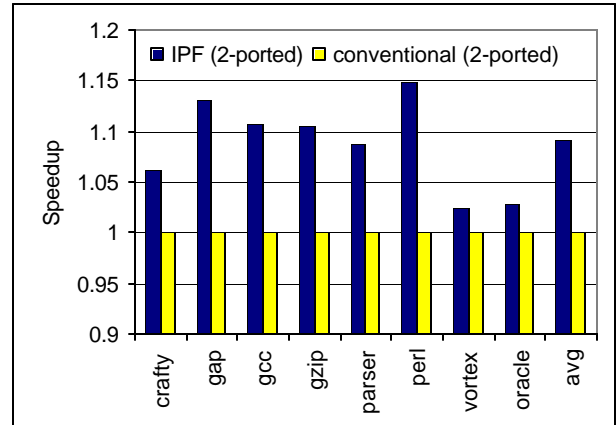


Figure 13 Load/Store Bandwidth Analysis (In-Order).

We also simulated the 2-ported configurations with an out-of-order model. Similar to the in-order model, the performance advantage of the register stack increases when the bandwidth of the L1D is limited. The performance advantage grows from 10.2% (4-ported) to 12.4% (2-ported) on average.

The performance advantage of the register stack is seen in both in-order and out-of-order machine models. The major contributor to the overall speedup is the reduced instruction count. The size of the register stack does not significantly impact the miss behavior of the L1D. However, the bandwidth of the L1D does have impact on non-register stack configurations. The out-of-order model consistently has bigger impact from the register stack. The following section explains this result.

7. Vitality of Stack Loads

[18][19][20] introduced the notion of load importance or criticality. [17] defined the concept of load *vitality*. Loads were classified into two categories: vital and non-vital. Vital loads are loads that must be executed as quickly as possible in order to maximize performance. Non-vital loads are loads that are less vital to program execution and for various reasons do not require immediate execution in order to maximize overall performance.

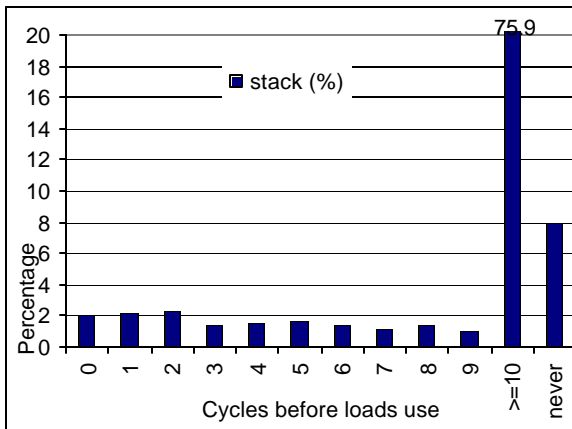


Figure 14 Dependence Distance of Stack Loads (In-Order).

We measure vitality of the stack loads by counting the number of cycles between the execution of the stack load and the execution of the earliest dependent instruction. Figure 14 shows vitality for stack loads for the in-order conventional configuration. The x-axis represents the number of cycles between the load finishing execution and the earliest dependent instruction using its data. The “0” bar (first bar) represents the % of vital instructions. For example, 2.0% of the stack loads’ data are used immediately and hence these are considered vital. The ≥ 10 bar represents data which are used in 10 cycles or greater, and the data which are never used are collected in the *never* category. We present the vitality for the conventional configuration in order to analyze the stack loads that the register stack successfully removes. As we can see, the stack loads are mostly non-vital (98% are non-vital). Therefore, the cache latency of stack loads is not a limiter in terms of performance. The loads that are eliminated by the register stack are mostly non-vital in the context of an in-order model.

Figure 15 shows the vitality of stack loads for an out-of-order model. Unlike the in-order model, 10% of stack loads are now vital. Section 6 shows that the performance advantage of the register stack is greater for an out-of-order model. More stack loads are now vital to performance because the out-of-order machine has moved up the dependent instructions. The register stack eliminates these vital stack loads in the out-of-order model and hence the IPF configuration outperforms the conventional configuration by a larger degree.

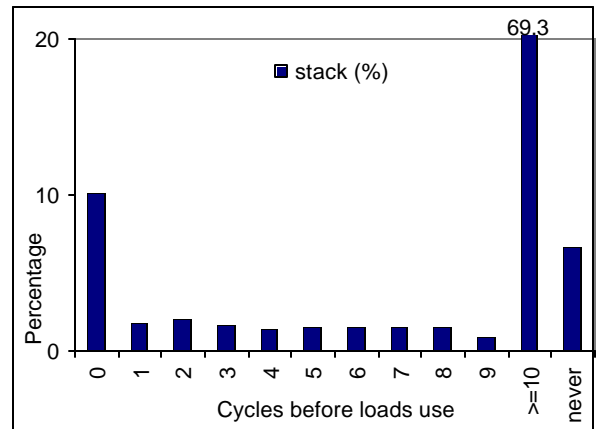


Figure 15 Dependence Distance of Stack Loads (Out-Of-Order Machine).

8. Conclusion

The goal of this work is to assess how well this technique actually performs for IPF machines and to explain why. In this paper, we examine the effects of the Itanium register stack on dynamic instruction counts, data cache miss rates, program execution time, and load instruction vitality. We conclude that the Itanium register stack with 96 physical stacked registers is successful in reducing the number of stack loads and stores by at least a factor of three compared to a conventional architecture. Furthermore, in the absence of the register stack, the Itanium architecture would have experienced a 31% increase in overall load traffic and 68% increase in overall store traffic. We show the register stack provides a 7%-8.3% and 10.2%-12% performance advantage in in-order and out-of-order machines, respectively. The higher performance advantage on an out-of-order machine is due to a greater number of stack loads becoming vital. Important benchmarks, such as *gcc* and *Oracle* have a large register window requirement. By making the physical register stack size larger than the current 96, almost all the spills and fills can be eliminated, resulting in even greater performance advantage. We believe that the register stack of the Itanium architecture is a very good feature, which will become more and more important as future workloads become more object oriented. Our preliminary measurements of Java workloads indicate that such workloads stress the register stack much more than the benchmarks in this study. Our future study will analyze such workloads.

9. References

- [1] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, R. Zahir, "Introducing the IA-64 Architecture", *IEEE Micro*, Sept-Oct 2000.
- [2] R. Zahir, J. Ross, D. Morris, D. Hess, "OS and Compiler Considerations In the Design of the IA-64 Architecture", *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.
- [3] Intel® Itanium™ Architecture Software Developer's Manual.
<http://developer.intel.com/design/itanium/manuals/index.htm>.
- [4] Y. Choi, A. Knies, L. Gerke, T. Ngai, "The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel Itanium Processor", *Proceedings of the 34rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2001.
- [5] D. Patterson, C. Sequin, "RISC I", *Conference Proceedings of the Eighth Annual Symposium on Computer Architecture*, May 1981.
- [6] T. Stanley, R. Wedig, "A Performance Analysis of Automatically Managed Top of Stack Buffers", *the 14th Annual International Symposium on Computer Architecture*, June 1987.
- [7] R. Cmelik, S. Kong, D. Ditzel, E. Kelly, "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks", *ACM SIGARCH Computer Architecture News, Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, Volume 19, Issue 2.
- [8] B. Furht, "A RISC Architecture with Two-Size, Overlapping Register Windows", *IEEE Micro*, Volume: 8 Issue: 2, April 1988, Pages 67-80.
- [9] D. Wall, "Register Windows Vs. Register Allocation", *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988.
- [10] M. Annavaram, T. Diep and J. Shen, "Branch Behavior of a Commercial OLTP Workload on Intel IA32 Processors", *To appear in Proceedings of the International Conference on Computer Design*, September 2002.
- [11] P. Wang, H. Wang, R. Kling, K. Ramakrishnan, J. Shen, "Register Renaming and Scheduling for Dynamic Execution of Predicated Code", *In 7th HPCA*, Jan 2001.
- [12] M. D. Hill, "A Case for Direct-Mapped Caches", *IEEE Computer*, pp. 25 - 40, Dec. 1988.
- [13] R. Kessler, R. Jooss, A. Lebeck, and M. Hill, "Inexpensive implementations of set-associativity", *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 17(3):131--139, 1989.
- [14] A. Seznec, "A case for two-way skewed-associative caches", *In 20th Annual International Symposium on computer Architecture*.
- [15] L. John and A. Subramania, "Design and performance evaluation of a cache assist to implement selective caching", *In International Conference on Computer Design*.
- [16] S. Cho, P. Yew, and G. Lee, "Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor", *Proc. of the 26th Int'l Symp. on Computer Architecture*.
- [17] R. Rakvic, B. Black, D. Limaye, and John P. Shen, "Non-vital Loads", *Proc. of the 8th International Conference on High-Performance Computer Architecture*, Feb. 2002.
- [18] S. Srinivasan and A. Lebeck, "Load latency tolerance in dynamically scheduled processors.", *in Proceedings of the Thirty-First International Symposium on Microarchitecture*, pp. 148--159, 1998.
- [19] E. Tune, D. Liang, D. Tullsen, B. Calder, "Dynamic Prediction of Critical Path Instructions", *In the Proceedings of the 7th High Performance of Computer Architecture*.
- [20] B. A. Fields, S. Rubin and R. Bodik, "Focusing Processor Policies via Critical-Path Prediction", *In proceedings of 28th International Symposium on Computer Architecture*.
- [21] R.D.Weldon, S. Chang, H.Wang, G. Hoflehner, P. Wang, and J. P. Shen, "Quantitative Evaluation of the Register Stack Engine and Optimizations for Future Itanium Processors", *In Interact-6 held in conjunction with the 8th International Conference on High-Performance Computer Architecture*, Feb. 2002.
- [22] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, H. Wang, "SoftSDV: A Pre-silicon Software Development Environment for the IA-64 Architecture", *Intel Technology Journal*, Q4 1999,
http://www.intel.com/technology/itj/q41999/articles/art_2.htm
- [23] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, "Simics: A Full System Simulation Platform", *Computer*, Volume 35, Issue 2, Feb. 2002, Pages 50-58.