Compiler Optimizations for Transaction Processing Workloads on Itanium® Linux Systems

Gerolf Hoflehner, Knud Kirkegaard, Rod Skinner,
Daniel Lavery, Yong-fong Lee, Wei Li

Intel® Compiler Lab
Santa Clara, California, USA

{gerolf.f.hoflehner, knud.j.kirkegaard, rod.skinner, daniel.m.lavery, yong-fong.lee, wei.li}@intel.com

Abstract

This paper discusses a repertoire of well-known and new compiler optimizations that help produce excellent server application performance and investigates their performance contributions. These optimizations combined produce a 40% speed-up in on-line transaction processing (OLTP) performance and have been implemented in the Intel C/C++ Itanium compiler. In particular, the paper presents compiler optimizations that take advantage of the Itanium register stack, proposes an enhanced Linux preemption model and demonstrates their performance potential for server applications.

1 Introduction

This paper describes compiler optimizations that help produce excellent server application performance and investigates their performance contributions. The compiler optimizations combined produce a 40% speed-up in OLTP performance and have been implemented in the Intel C/ C++ Itanium compiler. The Oracle production database has been used to run on-line transaction processing (OLTP) workloads on four Itanium 2 processor systems running the Linux operating system.

Intel's compiler for the Itanium processor family incorporates classical compiler optimization techniques [12], profile-guided optimizations, and new techniques that have been designed specifically for the Itanium architecture [2][9]. However, additional work and tuning efforts in the compiler were necessary to tackle challenging OLTP workloads [4][10][13]. This paper describes compiler optimizations that help improve OLTP workload performance and analyzes their performance impact.

A number of studies investigated the behavior of on-line transaction processing (OLTP) workloads. It is well known that a large instruction and data footprint as well as high I/O traffic characterize OLTP workloads [4]. Some papers investigate specific compiler optimizations like code layout optimizations and demonstrate that they are useful in reducing I-cache misses [13].

This paper takes a holistic view of the OLTP optimization problem. The substantial performance gains from the compiler are the result of utilizing a broad repertoire of compiler optimizations that exploit source code characteristics of the database code and utilize unique features of the Itanium architecture like the register stack engine (RSE) [5].

1.1 Contributions

This paper makes the following contributions:

- Discussions and measurements of compiler optimizations that make a difference for OLTP workload performance on a four Itanium 2 processor (1.5 GHz, 6M L3 cache) system running Oracle on a version of the Red Hat® Linux operating system.
- Discusses a new method to reduce the setjmp()/longjmp() call overhead.
- Proposes an enhanced Linux preemption model and discusses its performance potential for enterprise applications.

1.2 Organization of the paper

The rest of the paper is organized as follows. Section 2 describes compiler optimizations that helped improve performance of OLTP workloads. Section 3 shows the performance impact of the optimizations. Section 4 discusses key learnings and section 5 has concluding remarks and future work.

2 A repertoire of compiler optimizations for server applications

The performance barriers for an OLTP workload on an Itanium 2 system are D-cache, I-cache and ITLB misses and the memory traffic triggered by the register stack engine (RSE) [5]. This paper describes an optimization to reduce the RSE memory traffic in section 2.1, optimizations that are geared towards reducing I-cache and ITLB misses in sections 2.2 - 2.4, and optimizations that attempt to improve D-cache behavior in sections 2.5 - 2.8.

2.1 RSE traffic reduction

The Itanium architecture has 128 integer registers r0-r127. The upper 96 registers, r32-r127, are stacked. Each procedure can have its own variable size register stack frame of up to 96 registers. The stacked registers within a procedure are referenced as *architectural* registers. The hardware maps them to a micro-architecture dependent number of *physical* registers. For example, the first incoming parameter register in a procedure is referenced as r32. But this could be any physical register from r32 to the number of stacked registers implemented in the micro architecture. With the *alloc* instruction [5], the code generator explicitly specifies a procedure's register stack frame: the number of incoming parameters (i), the number of local (within the procedure) registers (l) and the number of outgoing parameters (o). The total number of registers in the register stack for the procedure is i+l+o <= 96. The parameter registers overlap for the caller and the callee [5]. The register stack frame is similar to a memory stack frame, but is managed by the register stack engine (RSE), a processor state machine.

```
foo():
  alloc rx=0,90,0
1:
                        bar():
  call bar()            →alloc ry=0,50,0
                        2:
                          …
  alloc rz=0,90,0       ←return
3:
```

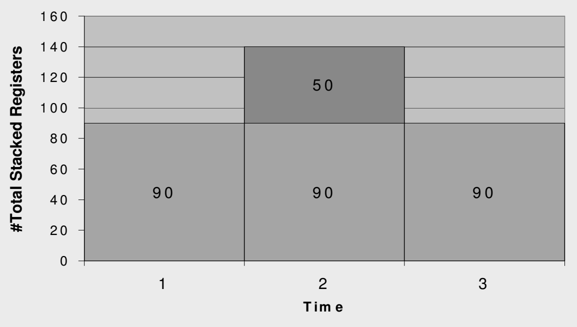

Figure 1. Unoptimized register stack usage

The Itanium® architecture allows an optimization that shrinks the register stack before a call site and restores it to its original size afterwards [14]. This technique can reduce the total number of registers consumed by the caller and callee and may result in a reduction of the overall RSE traffic for the application. Liveness analysis [12] determines the registers that are unused (or dead) at the point of the call. If the number of dead registers on top of the register stack exceeds a given threshold, the register stack is reduced by the amount of dead registers before the call. Parameter registers have to be remapped so that they stay on top of the resized register stack. In the examples in this section the parameter registers are ignored for simplicity.

Figure 1 shows assembly snippets of a function foo() calling a function bar() and snapshots of the register stacks at 3 points in time: after the allocation of 90 stacked registers in foo (1:), after the additional allocation of 50 stacked registers in bar (2:) and after the return from bar (3:). Combined, foo() and bar() use 140 stacked registers. This would trigger the spilling and filling (RSE traffic) of 44 registers by the RSE.

```
foo():
alloc rx=0,90,0
1:
…
alloc rz=0,30,0
2:                       bar():
call bar()              →alloc ry=0,50,0
                         3:…
alloc rz=0,90,0         ←return
4:…
```

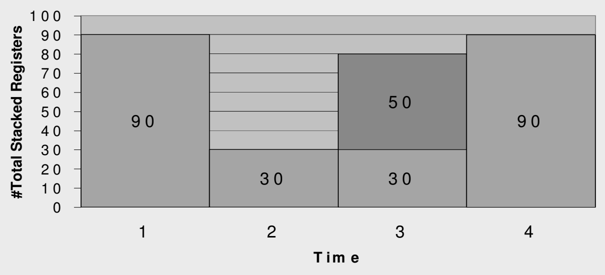

Figure 2. Extra alloc instructions and optimized register stack usage

In contrast, Figure 2 shows how the RSE traffic can be avoided. Assuming liveness analysis determines that 60 registers are unused (or "dead") on the register stack at the call of bar(), the compiler inserts an alloc instruction before the call to bar() to shrink the register stack frame to 30 registers (2:). The alloc instruction after the call to bar() allocates a stack frame of 50 registers for bar(). The combined register stack at this point (3:) holds only 80 registers. Finally, the alloc after the return from bar() re-sizes the stack frame of foo() to 90 registers (4:). Combined, foo() and bar() consume 90 registers, just like foo() alone, because bar() effectively reuses the stacked registers allocated by foo() to avoid potential RSE traffic.

This optimization is opportunistic in the sense that the compiler cannot have a perfect knowledge of the state of the RSE when it inserts the extra alloc instructions. Specifically, the compiler does not know if the reduction of the register stack will actually decrease the RSE traffic at run-time. On the other hand, the cost of the optimization is extra alloc instructions, which have scheduling constraints [5] and may contribute to an increase in code size. For an OLTP workload, the empirically found sweet spot was a threshold of 10 registers: alloc instructions are inserted only when at least 10 registers at the top of the register stack are found dead.

Restoring the register stack to its original size after the call is conservative and is an untapped optimization opportunity: the register stack after the calls only needs to be large enough so that it can fit the stacked register with the largest register number that is defined or used on any path from the point after the call to any return (or exit) block in the function [15].

2.2 Code scheduling and control speculation

OLTP workload performance is dominated by D-cache misses, so a small improvement in the number of scheduled cycles often does not help much. On the other hand, careful scheduling can help to reduce the impact of D-cache misses. An example is the use of control speculation. Control speculation can be used to move loads up past branches. This can increase instruction-level parallelism, especially when instructions that depend on the load are also moved up. However, when a use of a load is speculated, speculative cache miss stalls can occur. A speculative cache miss stall occurs when a use of a speculative load is stalled on a cache miss and the program execution does not continue on to the home block of the load. In this scenario the result of the load and its use is not needed and the processor stalls waiting for the load data unnecessarily. Loads can also be moved up to cover additional load latency in the event of a cache miss.

The Intel® compiler's global code scheduler has two heuristics to try to minimize the effects of cache misses. First, it does not speculate a load unless it meets a minimum threshold of usefulness as determined from the profile information. Speculating a load using ld.s or ld.sa requires the addition of a check instruction. The threshold provides a simple way to tune the speculation so that the gain from speculation outweighs the cost of the extra instructions. Second, at the –O1 optimization level, the compiler's heuristics are geared toward large instruction/data footprint applications. At this optimization level the compiler speculates loads, but does not speculate the uses of speculative loads. This avoids speculative cache miss stalls, covers additional load latency, and enables inline recovery code.

Figure 3 shows an example of inline recovery code. The compiler must generate recovery code to handle faults and exceptions for speculated loads. Recovery code re-executes the load (non-speculatively) and any dependent instructions. In the general case, recovery requires a check instruction and a branch to a recovery code block. For cases where only the load is speculated, inline recovery code uses the ld.sa and ld.c instructions instead of ld.s and chk.s, removing the need for a recovery code block. If a fault/exception occurs at the ld.sa instruction, the advanced load address table (ALAT [5]) entry associated with its result register (rx in the example) is cleared and the ld.c is executed. The number of ALAT entries is limited to 32 entries on the Itanium 2 processor. Therefore the implementation must be sensitive to the number of ld.sa lifetimes. The performance benefit is from decreased static and possibly dynamic code size (recovery code is expected to be infrequently executed).

With recovery code:	Inline recovery code:
`ld8.s rx=[ry]`	`ld8.sa rx=[ry]`
…	…
…	…
`chk.s rx, **rec**`	`ld.c rx=[ry]`
`L: …`	
`rec:`	
`ld8 rx=[ry]`	
`br L;`	

Figure 3. Example for inline recovery code.

2.3 Instruction prefetching

The Itanium® processor enables the compiler to control instruction prefetching by means of .few/.many completers on branch type instructions [5]. The completers specify how many bundles [5] get prefetched at the branch target. On Itanium, a bundle contains three instructions that can be executed in parallel. The machine fetches two bundles per cycles.

On our setup the heuristics that resulted in the biggest performance gain were to use the .many completer on all calls, returns and indirect branches, and on conditional branches when the probability – derived from feedback profile information – is greater than 50% that at least four bundles are executed at the target of the branch. The .few completer is used in all other cases.

2.4 Function layout optimizations

In this category, three optimizations contributed to performance: function inlining, function splitting and function grouping. All optimizations are effective when feedback-profiling information is available.

Function inlining removes the call overhead, improves code locality and increases the optimization scope, but it may come with the cost of an increase in code size. It was applied inter-procedurally to the hottest files of the application based on feedback profiling information.

Function grouping lays out the call graph based on the frequency of the call edges and reduces I-cache misses.

This is usually done during the inter-procedural optimization phase in the compiler. A simple heuristic that moves a function with close to zero entry code frequency into the cold section has been applied.

Function splitting partitions a function into hot and cold sections. It decreases ITLB misses by packing the hot parts of the functions together on fewer pages with the trade-off of long branches between the hot and the cold code sections. Similar to function grouping, a heuristic that moves basic blocks with close to zero frequency into the cold section gives a sizeable performance gain.

2.5 Data layout optimizations

The compiler performed two data layout optimizations. First, it is beneficial to move strings and constants to the read-only section instead of the data section. As the database starts multiple sessions, each process copies the (read/write) data section into its run-time image. Pushing data into the read-only section helps, because this reduces the size of the data sections and thus the copy overhead. The read-only data section is loaded only once at application start-up time. Thus it is better to move strings and constants to the read-only section instead of the data section. This is safe, when the application does not rely on modifying constants. Modifying constants is allowed by the K&R conventions of the C programming language [8], but is not allowed by ANSI C/C++ rules [1].

Second, the compiler implemented a set of heuristics for sorting the local data on the memory stack based on frequency and size. We observed that locality improves when data are allocated close to the stack pointer and size is used as a tiebreaker. Improved locality resulted in better cache line utilization. Pushing hot data structures closer to the top of the stack (close to the frame pointer) reduced D-cache misses in our setup.

2.6 Setjmp()/longjmp() optimizations

There are essentially two cases of setjmp() overhead reduction an Itanium® compiler can exploit. The setjmp() routine uses a buffer structure to save program state, which can be reinstated by a longjmp() call. Reducing the state necessary to save will result in fewer stores and loads at setjmp()/longjmp() invocations. Also, on the Itanium, it is incorrect to assign lifetimes that cross setjmp calls to stacked registers, unless the compiler can compute the correct interferences for these lifetimes.

2.6.1 Reduction of setjmp/longjmp() costs

Sequences of setjmp()/longjmp() code are a common pattern in database applications as well as other large system applications like OS kernels. A call to setjmp saves system state in an user supplied jmp_buf structure [1]. The return value of the setjmp in this case is zero. A longjmp call reinstates the function state from this buffer and execution control resumes after the setjmp. The return value after a setjmp in this case is non-zero. There is some similarity to context switches, and in both cases the compiler provides user options to reduce the context state and consequently the context-switch overhead.

Specifically, server applications (and integer code applications in general) don't need many floating-point operations and thus have only a small demand for floating-point registers. Because the Itanium® architecture has 128 floating-point (fp) registers, code that uses setjmp() intensively will benefit from limiting the number of fp registers available. In the compiler the user can direct the compiler to use only the eight scratch fp argument registers f8-f15. This has the following advantages: a) it reduces the number of stores or loads at the setjmp() call site when the program state is saved to or restored from the jmp_buf structure and b) it reduces the size of the memory stack of functions that define instances of jmp_buf structures. The tradeoff of restricting the number of floating-point registers in the compiler is a possible increase in load and stores due to spill code generated by the register allocator. Server applications are not floating-point intensive and we did not measure a performance loss from using only the scratch floating-point argument registers.

2.6.2 Utilizing the register stack for cross setjmp() lifetimes

Stacked registers are not saved in the jmp_buf structure. Thus, in the context of setjmp calls, lifetimes crossing setjmp call cannot simply be assigned stacked register on the Itanium architecture without special care. A lifetime is said to cross a setjmp() when it is *live* at the setjmp call. In the linear (acyclic) code example of Figure 4 the lifetimes for V1 and V2 don't overlap, so they do not interfere. Lifetime V1 is live across the setjmp call. The register allocator [2] in the compiler could assign the same stacked register r37 to both, V1 and V2. This situation is illustrated on the right hand side of Figure 4. Now assume that foo calls longjmp. After the return from the longjmp, setjmp returns a non-zero value, the comparison r==0 is false, but the value in r37, which is supposed to be V1 on this path (F), is wrong, because it has been overwritten earlier on the path (T) to foo() by the value of V2.

The solution to this problem in the compiler is to explicitly model the (otherwise implicit) control flow from any function that might call longjmp back to the associated setjmp call. The compiler adds a pseudo-edge to model the control flow in this case. Pseudo-edges represent the implicit control flow necessary for correct dataflow analysis. They are different from other control flow edges because neither can instructions be moved across them nor can compensation code blocks be inserted on them.

With this control flow graph enhancement, the register allocator can model interferences for cross setjmp() lifetimes like V1 correctly (Figure 5). Now V1 and V2 interfere and *cannot* be assigned the same register, and in particular not the same stacked register. So, when the compiler uses pseudo-edges to conservatively model interferences of cross setjmp() lifetimes, then stacked registers can be assigned to them. To enable this usage of stacked registers, the implementation of the longjmp() library routine must restore the register stack of the function containing the corresponding setjmp call.

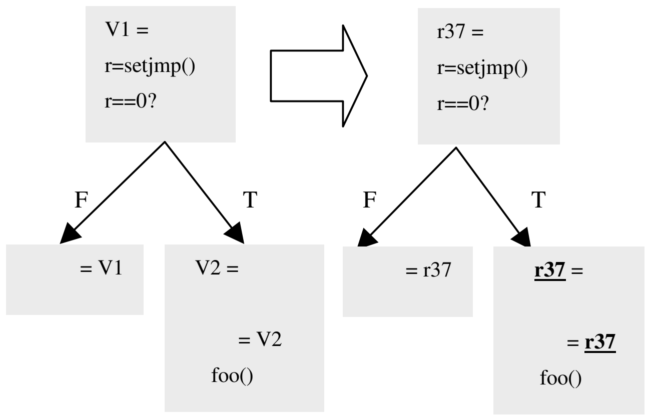

Figure 4. Lifetimes for V1 and V2 are assigned the same stacked register r37

Utilizing stacked registers for cross setjmp() lifetimes can reduce the spills/fills in functions with high register pressure. It also reduces the memory stack size (fewer spill locations on the stack) and the code size (fewer load/stores). It may also eliminate the spill/fill code for the callee preserved integer registers (r4-r7) at function entries/exits, because all 96 stacked registers become available for cross setjmp() lifetimes. The obvious cost of this optimization is an increase of RSE traffic: on a scaled setup we measured an increase of about 40% for the RSE traffic, but the gains from reducing the overall memory traffic and the code size did by far outweigh the loss. A more advanced optimization for the register stack (section 2.1) could perhaps reduce the RSE traffic increase.

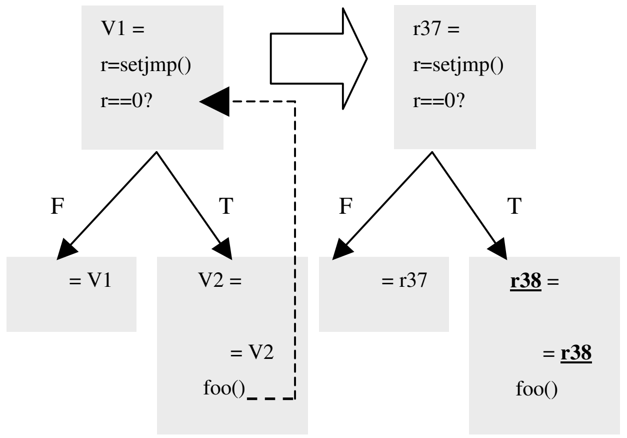

Figure 5. Adding pseudo-edges to catch cross-setjmp() lifetime interferences

2.7 Linux preemption model

On Linux the default application model and the generic ELF ABI [11] require the compiler to generate code that allows for possible symbol preemption of global symbols. Thus the generated code must also be position independent. In the example in Figure 6 the definition of the function foo() and the global variable g used in the shared library get preempted by default when the shared library is loaded. As a consequence the call to foo in bar() actually executes the user code. Relatively few applications take advantage of these features, and support for these features can cause significant run-time overhead. Therefore several performance opportunities are available to applications that don't require position independent code and the default symbol preemption model described in the ELF ABI. For example, a function can be inlined when it is not preemptible. So in the specific example of the shared library code in Figure 6 it is not legal to inline the call to foo in bar(). We have worked on developing safe software convention models that rely on the ELF visibility attributes and user input telling the compiler that position independent code is not required for the main executable.

user code:	shared library code:

`int g= 0;`
`foo(){`
`printf("User\n");`
`printf("g=%d\n",g);`
`}`
`main() { bar(); }`
	`int g= 2;`
	`foo(){`
	`printf("sh_lib\n");`
	`printf("g=%d\n",g);`
	`}`
	`bar() { foo();}`

Figure 6. Example for symbol preemption

By default global variables must be placed in the data section of the object file and accessed through the global linkage table [7] to ensure symbol preemption. This causes two levels of indirection to load a global scalar variable:

```
add r3 = @ltoff(data),gp
ld8 r2 = [r3]
ld4 r8 = [r2]
```

But the user can tell the compiler and linker that a symbol cannot be preempted. This allows the linker to put the symbol into the short data section. Specifically, the ELF visibility *protected* on a global symbol says that the symbol will be bound at link time. Then the global symbol can be loaded directly from the short data section:

```
add r2 = @gprel(data),gp
ld4 r8 = [r2]
```

Furthermore, it is possible that the object does not require position independent code generated for symbols that have ELF visibility other than *default*. This happens when the object will be linked with the main program, is not used in a shared object, and is loaded at a fixed address.

This can potentially avoid extra data cache misses, because the compiler can use the absolute symbol address and addressing does not need access the linkage table:

```
        movl r2 = ltconst(data)
        ld4 r8 = [r2]
```

There are other benefits of using the ELF visibility attributes also. For example, calls to functions that are *protected* do not have to save and restore the gp register [7]. In this case, the compiler can conclude that the call address will be bound at link time and cannot be preempted The user can also mark symbols that can be guaranteed to be resolved to shared objects. These symbols require function import stub [7], which is usually inserted by the linker in the plt (procedure linkage table) section of the generated object. When the compiler is informed that the symbol is in a shared object, it can inline the stub, which improves code locality.

2.8 Memory Ordering

Server code usually has a significant number of volatile data references. The volatile qualifier is used to force loads and stores of the data object to stack memory [1]. The default model in the compiler forces ordered volatile data accesses, but might be too restrictive and can be relaxed. For ordered data accesses Itanium® has .rel/.acq completers for release/acquire data accesses) on load/stores [5]. *Release* data accesses guarantee that all previous data accesses are visible, *acquire* data accesses guarantee that they are made visible before any subsequent data accesses. Softening the memory ordering semantics may give performance benefits. Thus the compiler has an option to support unordered volatile data accesses. Specifically, under an option the *.rel/.acq* completers[5] will not be issued on referencing loads and stores of volatile data. However, as some memory accesses have to be ordered, the compiler does provide ordered load and store intrinsics. The intrinsics have a cost also: they do require source code changes and therefore, in the scenario described above, they do require an intimate knowledge of the source code. On the other hand, usually only a few source code lines have to be changed.

3 Evaluation of compiler optimizations

In a performance development environment geared towards measuring OLTP workloads, a scientific evaluation of the compiler optimizations is difficult. There is ongoing change in the environment: the OS version, the database code, the compiler and the hardware change constantly. Sometimes instabilities infiltrate the test system resulting in an unacceptable run-to-run variation of several percent. The results are collected on different configurations and changing systems. Therefore reproducing the results exactly may be difficult.

Measurements were done on both small-scale cached OLTP setups and much larger scaled setups. Cached setups significantly scale down the database size so that the working set fits within system memory, resulting in negligible disk I/O. Essentially, on a cached setup an OLTP workload runs CPU bound and compiler optimizations usually have a higher impact on OLTP performance compared to scaled setups. Scaled setups are expensive, challenging to configure, have a large number of disks and are less amenable to compiler optimizations: a high speed-up on a cached system does not necessarily translate into a high speed-up on a scaled setup. Sometimes measured gains on cached system simply evaporate on a scaled setup, most notably for improvements from code scheduling. In our experience, however, in most of the cases the measurement on a cached system was in the ballpark of what was measured on a scaled system. In general, the run-to-run variation was very low (<0.3%) for both, scaled and cached systems, which allowed for on-going empirical measurements. When instabilities have been discovered, measurements have been redone on a stabilized system. The performance numbers in this paper are from scaled runs. When an optimization was measured several times, the lowest number measured is presented.

Profile data was collected with an instrumented database binary during a steady state phase of an OLTP workload run on a system similar to a scaled setup. Before feedback collection started, a short run warmed up the caches.

For some optimizations this section provides additional performance data for contrast, for example for tests in the SPECint2000 benchmark suite.

3.1 Measurement framework

We used version 7.1 of the Intel® C/C++ compiler, Red Hat® Linux Enterprise Edition 2.1 and 3.0, versions of the Oracle® database source code and measured OLTP workloads on two different scaled systems, each equipped with four Itanium 2 processors and 32 GB of memory. The L3 cache size was 3M and 6M respectively. The corresponding cached systems used 8 GB of memory.

All the measurements reflect each compiler optimization at one specific point in time. The measurements reported here were performed over a course of more than 12 months, during which intensive analysis and optimization were conducted. We tested one optimization at a time and measured its performance impact. Only optimizations that showed gains on a cached setup were re-evaluated on a scaled setup. There, Itanium performance monitors [5] were used to measure the effects of an optimization on D-cache, I-cache and the RSE traffic.

3.2 Performance monitor data

In Table 1 we chose a representative subset of the optimizations discussed in this paper. For each optimization the performance counter data for L1 and L3 D-cache misses as well as the L3 D-cache miss ratio are listed.

Table 1.Performance monitor data

	Optimizations				
Metrics	**RSE Optimization**	**Read-Only Data**	**Small setjmp() buf**	**setjmp() registerization**	**Linux preemption**
Throughput	1.15%	1.64%	1.08%	1.75%	8.22%
L1DRead Miss	-3.41%	-2.18%	1.29%	10.91%	-13.35%
L3D Read Miss	-0.91%	-0.52%	1.17%	-4.00%	-19.38%
L3 MissRatio	-3.32%	2.38%	-5.14%	-2.49%	-3.33%
RSE spill	-4.15%	n/a	n/a	n/a	n/a
RSE fill	-4.04%	n/a	n/a	n/a	n/a

Both data and instructions reside in the L2 and L3 cache, but the performance monitors cannot count instruction and data misses separately for the L2 cache. So L2 data are less meaningful and have not been included. RSE data are not included for most optimizations. In the table throughput is #transaction/time and is the measure for speed-up. The L3 Miss Ratio is measured by #read misses / #reads.

The data compare a binary with an optimization enabled versus a binary that was compiled without the optimization. The RSE optimization described in section 2.1 reduced the RSE spills/fills by about 4%, improved the L3 miss ratio by more than 3% and increased throughput by 1.15%. Placing constants and strings in the read-only data section as described in section 2.5 reduces overall memory traffic and increased throughput by 1.64%. The setjmp() optimizations (section 2.6) improved the L3 miss ratio due to a smaller setjmp() buffer and a reduction in spills/fills respectively. The enhanced Linux preemption model produced the largest improvement: throughput is up by 8%, all cache miss data are down. As the data in the table show there does not seem to be a magic formula that relates performance monitor data to throughput. Also, the run-to-run variation in performance monitor data was higher than the run-to-run variation for a single workload.

3.3 Overview of compiler optimizations speed-ups

The compiler optimizations for OLTP workloads are divided into six classes: RSE optimization, Data Layout (including local data sorting and read-only data section usage), Optimizations for setjmp() (reduction of the setjmp() buffer, Registerization of cross setjmp() lifetimes), Code Optimizations (including inline recovery code, function splitting, speculation for latency, selective instruction prefetch hints, function inlining and function grouping), enhanced Linux preemption model and dynamic feedback profiling.

In Figure 7, on the x-axis is the optimization class, on the y-axis is the speed-up measured for all the optimization in that class. Again, the speed-up was measured as percentage improvement in #transactions/time (throughput) on a scaled setup. For example, the measured performance gain from the setjmp() optimizations (section 2.6) is 2.85%. The combined speed-up from all optimizations described in the previous chapter is 21.88%. In addition, enabling dynamic feedback profiling initially, before any other optimization described in this paper was turned on, did give a speed-up of about 20%. A hot function in database code might be considered cold or insignificant in a SPECint20000 benchmark. However, although the profile data is flat, there are hot paths within the functions. This enables compiler optimizations to be more effective in addition to the well-known improvements from profile-driven function and block ordering. Combined, all compiler optimizations improve the throughput of OLTP workloads by more than 40%.

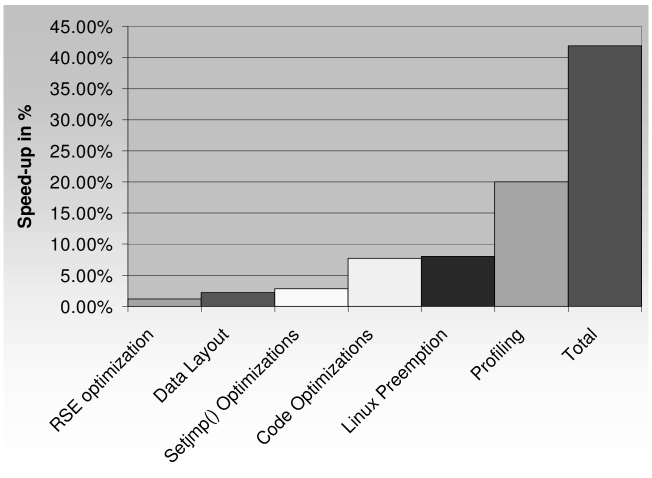

Figure 7. Speed-ups per optimization class

3.4 Speed-up from reducing D-cache stall cycles

Figure 8 shows the speed-ups from data access optimizations. The biggest impact is from enhancing the Linux preemption model to reduce the number of preemptible data and function symbols (section 2.7). The performance gains from using software convention models for symbol preemption other than the Linux default model are from reducing the L3 D-cache misses and have two sources. First, a protected (non-preemptible) symbol can be put into the short data section and be directly accessed there (gp-relative addressing). Second, for protected and position-dependant symbols absolute addressing saves also one level of indirection compared to the ltoff addressing mode. The software convention models we developed can speed-up all applications with significant global data traffic. On SPECint2000, 176.gcc, 186.crafty, 253.perlbmk, 254.gap and 255.vortex gained 7%, 15%, 11%, 4% and 3% respectively. The enhanced Linux preemption model also removes most of the save/restores of the gp (global pointer, [7]) register before and after function calls, but this improvement was measured to be in the noise for OLTP workloads and SPECint2000 benchmarks.

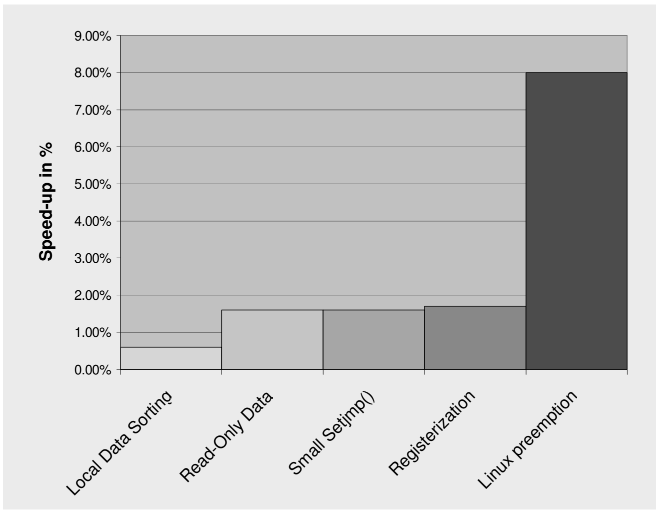

Figure 8. Speed-ups of data access optimizations

3.5 Speed-up from reducing D-cache stall cycles

Registerization of cross setjmp() lifetimes reduces the spills/fills in functions with high register pressure. It also reduces the memory stack size (fewer spill locations on the stack) and the code size (fewer load/stores). A side effect of this optimization is that it in some functions it eliminates the spill/fill code for the callee preserved integer registers (r4-r7 on Itanium) at function entries/exits. On a scaled run, it was observed that registerization increases the RSE traffic by about 40%, but the overall speed-up from this optimization was 1.75%.

Restricting the floating-point register usage to only scratch (caller-save) registers did not result in extra spill code, because there is not much floating-point code in the database source. A smaller setjmp() buffer was enabled by using fewer floating-point registers (by removing the 19 preserved floating-point register f2-f5, f16-f31 from the buffer), which resulted in fewer store/load to/from the setjmp() buffer.

The setjmp() optimizations combined have been measured to give gains between 2-5% for other database-like applications as well. They have no impact on tests of SPECint2000 benchmark suite, as all the longjmp() calls there are cold.

It is beneficial to move strings and constants to the read-only section instead of the data section. As the database starts multiple sessions, each process copies the (read/write) data section into its run-time image. Pushing data into the read-only section helps, because this reduces the size of the data sections and thus the copy overhead. The read-only data section is loaded only once at application start-up time.

Sorting local data based on frequency and size gave a small performance gain of 0.6% on a scaled setup as it reduced the L3 cache misses. The layout of local data close to the top of stack improved cache locality for the database application in our setup.

3.6 Speed-up from reducing I-cache stall cycles and instruction latencies

Figure 9 shows the performance speed-up from code optimizations that target better I-cache utilization or help hide load latencies.

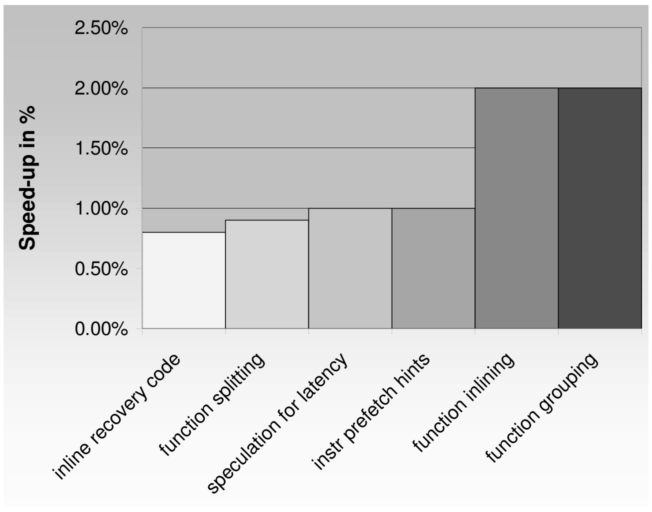

Figure 9. Speed-ups of code layout optimizations

Function grouping for close to zero frequency functions has been measured to give a speed-up of up to 3% on a cached setup and never lower than 2% on a scaled setup.

The data for function inlining on a small subset of the source base is also conservative in the sense that the 2% gain is the lowest measured on any measurement setup.

Selective instruction prefetch hints as described in section 2.3 increased the throughput by more than 1%.

Each of the above three optimizations decreased the number of L3 instruction references by about 10%.

Hiding D-cache latencies by avoiding control speculative cache miss stalls (section 2.2) improved performance by about 1%.

Function splitting reduced ITLB [5] misses and gave a gain of less than 1%, but consistently more than 0.5% on any scaled setup.

The 0.8% speed-up for inline recovery code is from a reduction in L3 cache misses.

The total contribution from this class of optimizations is 7.7%. Also, in this optimization class the performance gains measured on a cached setup were in general bigger than on a scaled setup. In one particular instance, for the relaxed memory ordering (section 2.8) a 3.6% speed-up had been measured on a cached system consistently in multiple runs. However, this gain did not materialize in later measurements on a scaled setup.

3.7 Speed-up from reducing RSE stall cycles

On OLTP workloads we measured a performance gain of 1.15%, when at least 10 registers are saved on the register stack before a call is invoked. This gain reflects upon that the database code demands more stacked registers and has a deeper call graph than, for example, the benchmarks in the SPECint2000 benchmark suite [14].

3.8 Interactions between optimizations

To get an idea about how well the speed-up numbers for the optimizations might add up, we did turn off all the optimizations except for dynamic feedback profiling. The performance difference between the two binaries was 18.76%, about 3% below the compound throughput gain of 21.88% for the optimizations in Figure 7. This indicates that there is negative interaction among the optimizations, which is an opportunity for more detailed analysis and research.

4 Key learnings

The analysis of compiler optimizations for OLTP workloads presented in this paper has yielded a number of interesting observations.

First, contrary to conventional wisdom, compiler optimizations can make a big difference for OLTP workloads on both cached and scaled setups.

Second, a large set of compiler optimizations is necessary to achieve excellent server application performance.

Third, optimizations that target memory traffic do scale very well and contribute significant gains to OLTP workload performance. Some of these optimizations are supported by the compiler, but also require changes to the source code or user options. Specific examples for this are the change of the jmp_buf structure for setjmp()/longjmp() calls and the support of safe software convention models for symbol preemption on the Linux operating system.

Fourth, dynamic feedback profiling is key to peak OLTP performance. A hot function in database code might be considered cold or insignificant in a SPECint2000 benchmark. However, although the profile is flat, there are hot paths within the functions and the database code does benefit from function layout and block ordering.

Fifth, some optimizations need to be tuned for memory intensive code. For example, in applications with a significant amount of D-cache misses it can be beneficial to speculate loads, but not to speculate the uses of a speculative load. This may avoid speculative miss stalls, cover additional load latency, and enable inline recovery code.

Sixth, the Itanium® register stack is very effective for OLTP workloads. The RSE optimization (section 2.1, [14]) in its simple implementation reduced the RSE traffic by more than 1%. Given that the total RSE traffic for OLTP workloads is less than 10% even after applying the setjmp() optimization described in section 2.6, a 1% gain is sizeable. OLTP workloads are both more call intensive and have bigger functions with higher register pressure than the SPECint2000 benchmarks suite, where the RSE traffic for the entire suite is only about 1% on a single Itanium 2 processor system with 6M L3 cache.

Finally, the register stack also enables the compiler to registerize lifetimes that cross setjmp calls (section 2.6). This can reduce the memory traffic for applications rich with setjmp calls. In this context the compiler introduced pseudo-edges to explicitly model the control flow from any function that might call longjmp back to the associated setjmp call. A similar technique can also be used to optimize C/C++ programs that use exception handling.

5 Concluding remarks and future work

The paper discussed a repertoire of well-known and new compiler optimizations and their impact on performance for on-line transaction processing (OLTP) workloads on four Itanium 2 processor systems running Oracle on a version of the Red Hat Linux operating system. These optimizations combined produce a 40% speed-up in OLTP performance. The optimizations and their performance contributions have been analyzed in detail. Also, both changes to the source code and enhancements of the Linux preemption model that make the compiler generate better code have been described.

Memory traffic continues to be the major bottleneck for OLTP workloads. More compiler research is needed to explore effective, scalable optimizations that can help reducing the memory traffic in large enterprise applications like databases.

The interaction among the compiler optimizations presented in the paper may well deserve further study. We also expect more performance gains from enhancing and tuning some of the methods discussed.

6 Acknowledgments

Many people contributed to the OLTP performance effort on the Itanium processor family. In particular, we would like to thank Pierre Calixte, Michael Chynoweth, Rajiv Deodhar, Corey Gough, Rakesh Krishnaiyer, Darla Kuras, Chu-chow Lim, Chinang Ma, Bernhard Mulder, Hubert Nueckel, Terry Pricket, Hiremane Radhakrishna, Robyn Sampson, Stephen Skedzielewski, Sharad Tripathi, Peng Tu, Vish Viswanathan and David Zhuang.

The Oracle development and performance teams also contributed to performance analysis and suggested new compiler optimizations. In particular, we would like to thank Brian Hirano and Waleed Ojeil from Oracle Corporation.

Last, but not least, we thank the anonymous reviewers for providing valuable feedback.

7 References

[1] ANSI Standard Programming Language C, Committee Draft ANSI, January 1999

[2] J. Bharadwaj, W. Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, J. Pierce, "The Intel IA-64 Compiler Code Generator", IEEE Micro, Sept./Oct. 2000, pp 44-52

[3] G. Chaitin. “Register Allocation and Spilling via Graph Coloring”, Proc. of the SIGPLAN '82 Symp. on Compiler Construction, Vol. 17, No. 6, June 1982

[4] R. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, J. Shen, “Scaling and Characterizing Database Workloads: Bridging the Gap between Research and Practice”, Proc. of the 36th Annual ACM/IEEE Int. Symposium on Microarchitecture, MICRO 2003

[5] Intel Corporation, “IA-64 Application Architecture Software Developer's Manual”, Volumes I-IV, http://developer.intel.com (January 2000)

[6] Intel Corporation, "Itanium™ Processor Microarchitecture Reference", URL:ftp://download.intel.com/design/IA-64/Downloads/24547401.pdf (August 2000)

[7] Intel Corporation, “Itanium™ Software Conventions and Runtime Architecture Guide”, May 2001

[8] B. Kernighan, D. Ritchie, “The C Programming Language”, Prentice Hall, 1988

[9] 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, Nov 2000, pp 60-68.

[10] S. Leutenegger, D. Dias, “A Modeling Study of the TPC-C Benchmark”, Proc. of the 1993 ACM SIGMOD Int. Conference on Management of Data

[11] Linux specifications, http://www.linuxbase.org/spec/refspecs/LSB_1.3.0/IA64/spec.html

[12] S.Muchnick, Advanced Compiler Design and Implementation, Published by Morgan Kaufman (1997)

[13] A. Ramirez, L. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. Lowney, M. Valero, “Code Layout Optimizations for Transaction Processing Workloads”, Proc. of the 28th Annual Int. Symposium on Computer Architecture, 2001

[14] A. Settle, D. Connors, G. Hoflehner, D. Lavery, “Optimization for the Intel Itanium Architecture Register Stack”, Proc. of the International Symposium on Code Generation and Optimization, CGO 2003

[15] A. Settle, D. Connors, G. Hoflehner, D. Lavery, “Compiler Controlled Register Stack Management for the Intel Itanium Architecture”, EPIC-3 workshop, 2004

IEEE COMPUTER SOCIETY