Just-In-Time Java ™ Compilation for the Itanium® Processor

Tatiana Shpeisman
Intel Labs
Tatiana.Shpeisman@intel.com

Guei-Yuan Lueh
Intel China Research Center
Guei-Yuan.Lueh@intel.com

Ali-Reza Adl-Tabatabai
Intel Labs
Ali-Reza.Adl-Tabatabai@intel.com

Abstract

This paper describes a just-in-time (JIT) Java[1] compiler for the Intel® Itanium® processor. The Itanium processor is an example of an Explicitly Parallel Instruction Computing (EPIC) architecture and thus relies on aggressive and expensive compiler optimizations for performance. Static compilers for Itanium use aggressive global scheduling algorithms to extract instruction-level parallelism. In a JIT compiler, however, the additional overhead of such expensive optimizations may offset any gains from the improved code.

In this paper, we describe lightweight code generation techniques for generating efficient Itanium code. Our compiler relies on two basic methods to generate efficient code. First, the compiler uses inexpensive scheduling heuristics to model the Itanium microarchitecture. Second, the compiler uses the semantics of the Java virtual machine to extract instruction-level parallelism.

1. Introduction

This paper describes a just-in-time (JIT) Java byte code compiler for the Intel Itanium processor. The Itanium processor is a statically scheduled machine that can issue up to 6 instructions per cycle – its performance, therefore, depends on aggressive compiler techniques that extract instruction-level parallelism (ILP). Building a JIT compiler for this processor is challenging for two reasons. First, the time and space taken by the JIT is an overhead on program execution – applying aggressive (time and memory consuming) JIT optimizations to obtain the last few percentage of performance gain may not be justified because the additional compilation overhead will slow down application load time, which may offset any gains from optimizations. Second, a Java JIT must be reliable because it is partially responsible for enforcing security (for example, by inserting checks or unwinding stack frames) – aggressive ILP optimizations (such as predication, speculation, and global scheduling) are complicated and time consuming to implement reliably.

In this paper, we describe lightweight code generation techniques for generating efficient Itanium code from Java byte codes. We also describe potential pitfalls in Itanium code generation as well as optimizations that are of limited effectiveness. Our JIT compiler relies on several inexpensive techniques to generate efficient Itanium code. First, the compiler uses heuristics to model Itanium micro architecture features (e.g., bypass latencies and execution unit resources). Second, the compiler leverages the semantics and metadata of the Java Virtual Machine (JVM) [17] to extract ILP. Our experimental results show that the goal of inexpensive but effective optimizations on Itanium is achievable.

The JIT compiler is part of the Open Runtime Platform (ORP) [15], an open source JVM that uses the GNU classpath Java libraries [10]. We ported ORP to run on the Itanium Processor Family (IPF) but we have not yet tuned the performance of ORP's machine-specific and performance-critical components, such as synchronization, garbage collection, and exception handling. Figure 1 compares the performance of ORP with the IBM JDK v1.3 Beta 2 (which is also an IPF JVM) for several benchmarks from the Spec JVM98 benchmark suite [21]. These measurements were gathered on a 733 MHZ Itanium workstation. ORP is competitive with the IBM JDK for compute-intensive benchmarks such as 201_compress, 202_jess and 222_mpegaudio, all of which spend most of their execution time in JIT-compiled code. The IBM JDK outperforms ORP significantly for benchmarks that demand fast synchronization (227_mtrt and 209_db) and exception handling (228_jack).

The remainder of this paper is organized as follows. Section 2 presents an overview of the IPF architecture and the Itanium processor micro architecture. Section 3 describes the optimization phases of our JIT compiler.

Section 4 describes the register allocator, which is

[1] All third party trademarks, trade names, and other brands are the property of their respective owners

based on the linear scan register allocation algorithm [19] and enhancements that minimize the amount of information the compiler has to store to support garbage collection. We propose an improved linear scan allocator that performs register coalescing without increasing the asymptotic complexity of the allocator.

Section 5 describes the code scheduler. We show how the scheduler models the latencies and resource constraints of the Itanium processor, and performs memory disambiguation and safe speculation of memory loads by leveraging the rich metadata provided by the JVM.

Section 6 describes Itanium-specific optimizations that have only minor benefits (aggressive sign-extension optimization and predication) and describes the potential pitfall of ignoring branch hint bits on the Itanium processor.

2. The Itanium processor

The IPF architecture organizes instructions into static groups called *bundles* [13]. A bundle is 128 bits long and contains 3 instruction slots. Each instruction has a type (I, M, F, B, A, L) corresponding to the execution unit type on which the instruction will execute. The I-, M-, F- and B-type instructions can be executed only on integer, memory, floating-point or branch units, respectively. A-type (arithmetic) instructions, such as integer add, can be executed on either memory units or integer units. L-type instructions are for loading of 64-bit immediates and use the integer and floating-point units. L-type instructions take two slots in a bundle; all other instructions take one slot. Only certain combinations of instruction types are valid inside each bundle. There are 24 valid combinations each of which is called a *template*. Some templates contain *stop bits*, which indicate cycle breaks between instructions – these are necessary to enforce dependencies between instructions within and across bundles. The sequence of instructions between stop bits or between a stop bit and a taken branch is called an *instruction group*.

The IPF architecture is a load-store architecture and has only the register indirect addressing mode for data access instructions. The register file contains 128 integer, 128 floating-point, 64 predicate, and 8 branch registers. The integer registers are divided into 96 stack registers and 32 static registers. Each procedure starts by allocating a register frame using the alloc instruction. The Register Stack Engine (RSE) ensures that the requested number of registers is available to the method by moving registers between the register stack and the backing store in memory without explicit program intervention.

The IPF architecture is a 64-bit architecture and provides limited support for 32-bit integer arithmetic. Load instructions zero-extend 8-, 16- and 32-bit integer values. The compiler must generate explicit sign-extension instructions whenever the upper 32 sign bits may affect program semantics.

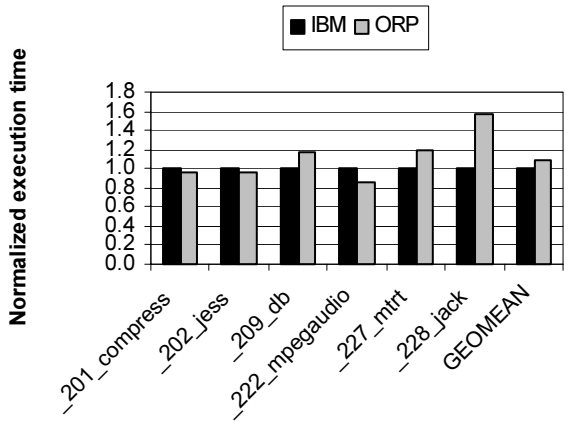

Figure 1. Performance comparison of ORP versus IBM JDK.

One of the salient features of the IPF architecture is predication. Most instructions have a qualifying predicate – if the predicate is evaluated to true, the processor commits the instruction's side effects to the architectural state; otherwise, the processor discards the instruction's side effects. Predication is used to remove branches, thus reducing branch misprediction penalties, increasing the scope for instruction scheduling, and possibly reducing path lengths.

Another feature of the IPF architecture is branch prediction hint bits, which allow a compiler to specify what kind of branch prediction should be done for a branch instruction. The branch prediction hint can be one of four values: statically taken, statically not taken, dynamically taken, and dynamically not taken. The usage of the hint is micro architecture specific.

The Itanium processor [14] is an implementation of the IPF architecture. This processor can decode two bundles per cycle and issue up to six instructions per cycle. The micro architecture has nine execution units: two memory, two integer, two floating-point and three branch units. NOP instructions are assigned to execution units and are executed like any other instruction. The micro architecture will stall if the instructions inside an instruction group oversubscribe the processor's execution unit resources. The execution units have varying execution latencies and there are non-unit bypass latencies between certain execution units.

3. Compiler overview

There are two kinds of compilation infrastructures that are commonly adopted in JVM implementation: mixed mode and compilation mode. The former comprises of an

interpreter that interprets cold (infrequently-executed) methods and a compiler that compiles identified hot (frequently-executed) methods [23][22]. The later compiles methods using either multiple JIT compilers [8] or multi-level compilation [4]. The infrastructure we use is compilation mode, which is similar to JUDO [8]. The scope of this paper focuses solely on lightweight code generation techniques that can be used in fast code generation [8] and 1-level optimization [22]. Figure 2 shows the structure of the JIT compiler. The compiler consists of two major components: the front end and the back end. The front end builds the intermediate representation (IR) and performs machine-independent optimizations. The back end lowers the IR to the machine level and performs architecture-dependent optimizations such as register allocation and code scheduling.

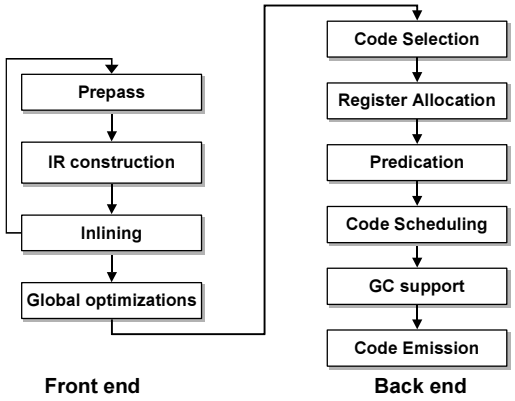

Figure 2. Compiler structure.

The compiler front end is similar to the one described in [8]. Compilation starts with the prepass phase, which traverses the Java bytecodes and collects information such as basic block boundaries and the Java operand stack depth. The IR construction phase then uses this information to build the control-flow graph and IR instruction sequences for each basic block. It also performs local common subexpression elimination across extended basic blocks.

The inlining phase identifies call sites that are candidates for inlining. The front end repeats the IR construction phase for the inlined call sites, merging the inlined method's IR into the caller's IR. The inlining policy is based on static heuristics that guard against code explosion by limiting the size of the inlined method and the total size of the method after inlining. When inlining a virtual method that can be overridden, the compiler generates a guard that tests whether the inlined method is the right method to be invoked. This guard branches to the normal method invocation code sequence if the test fails. The last phase of the front end is the global optimization phase, which performs copy propagation, constant folding, dead code elimination [2], and null pointer check elimination.

The first phase of the compiler back end is code selection, which lowers high-level operations (such as field or array element accesses) to IPF code sequences. Global register allocation assigns physical registers, generates spill code, and performs coalescing to eliminate register moves. Predication [3] eliminates some branches by predicating the instructions that are control dependent on a branch with the branch condition.

Code scheduling assigns instructions to execution cycles, packs them into the bundles, and selects templates. The scheduler must insert stop bits so that the hardware respects data dependence hazards. It must also use NOP instructions to fill unused instruction slots inside a template. The backend performs code scheduling after register allocation so that it also schedules any spill code generated by register allocation and so that move operations that are eliminated by register coalescing do not interfere with scheduling.

The GC support phase computes the set of live object references at every instruction [20] and writes this information into a compressed GC map data structure. The compiler uses this information at run-time to report the root set of live references to the garbage collector. The final code emission phase emits the native IPF binary code into memory for execution.

4. Register allocation

Choosing a register allocation algorithm involves making a trade-off between compilation time and the quality of the generated code. Graph coloring [5][6] is a commonly used approach to assign registers, which requires building the interference graph. Given the large number of registers available on the Itanium processor and the RSE a simpler algorithm is sufficient to yield a good register allocation. To favor fast compilation time, we chose the linear scan register allocation algorithm [19] that has linear-time complexity in assigning registers and does not require building the interference graph.

Linear scan register allocation algorithm approximates live ranges of variables using live intervals. A live interval is a contiguous set of instructions that includes the variable's live range.

The register allocator first creates a linear ordering of the instructions according to a traversal of the control flow graph and then builds a list of live intervals sorted by the order of the first instruction of the interval. The interval size has a direct influence on the quality of register allocation because the longer an interval, the more likely it overlaps with other live intervals. The register allocator tries to minimize interval lengths by traversing the blocks in topological order. For a method with `I` instructions, `V`

virtual registers, and B basic blocks, the live intervals are computed in O(I+VB) time given precomputed liveness information.

After computing the live intervals, the register allocator assigns registers to live intervals in a single pass over the sorted list of live intervals in O(V) time. To avoid excessive RSE activity, the register allocator tries to assign the minimum number of physical registers. It also tries to avoid spilling variables and minimize anti dependencies, both of which hinder code scheduling. The register allocator starts with a set of four available registers. It assigns registers from this available set in a round-robin fashion. If it runs out of registers, it adds another register to the available set or spills if it cannot add another register to the set. This strategy avoids superfluous anti dependencies while minimizing the number of used registers and the number of spills.

4.1. Register coalescing

Register coalescing eliminates moves by assigning the same register to the source and destination operands of a move. Coalescing is an important optimization for Itanium because the extra move instructions take up execution resources during scheduling and add to critical path lengths, especially floating-point moves, which have a latency of five cycles.

If the live ranges of the source and the destination of a move instruction do not interfere the compiler can coalesce the live ranges and eliminate the move [5]. For example, Figure 3 shows that variables v, t1 and t2 can be assigned the same physical register and the two move instructions can be eliminated. Linear scan register allocation does not compute the live ranges of variables and can coalesce two variables only if their live intervals do not intersect. As a result, it easily misses register coalescing opportunities. For example, Figure 3b shows that v and t2 cannot be coalesced because their intervals overlap.

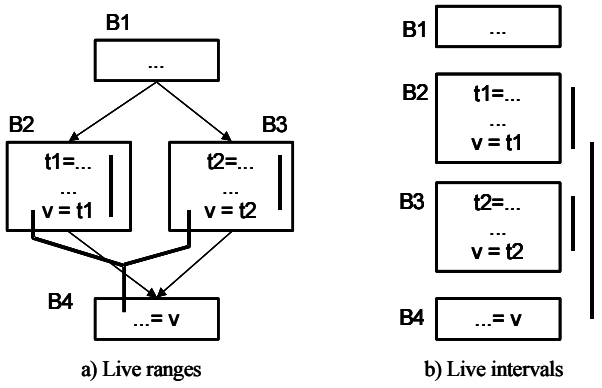

Figure 3. Live interval as approximation of live range

The JIT compiler performs register coalescing as a separate optimization inside linear scan register allocation. Rather than relying on the limited opportunities for live interval coalescing, the compiler attempts to coalesce a source and destination of a register move instruction if the live interval of the source does not intersect with the live range of the destination. The algorithm uses the live interval information already computed for the linear register scan and computes the live ranges on the fly during a single reverse pass over the instructions.

The register coalescing algorithm iterates over all the instructions in reverse order. When it encounters a register move instruction d = s, it checks whether this instruction is the end of the live interval of the variable s. If so, it marks d and s as candidates for coalescing. If the algorithm reaches the beginning of the live interval of s without finding a conflict, it coalesces d and s. A conflict occurs if d is live inside the live interval of s; that is, if the live interval of s contains a use or definition of d, or d is live at the basic block boundary inside the live interval of s. This algorithm is powerful enough to coalesce multiple variables. For example, in the program fragment shown in Figure 3, it coalesces v with both t1 and t2.

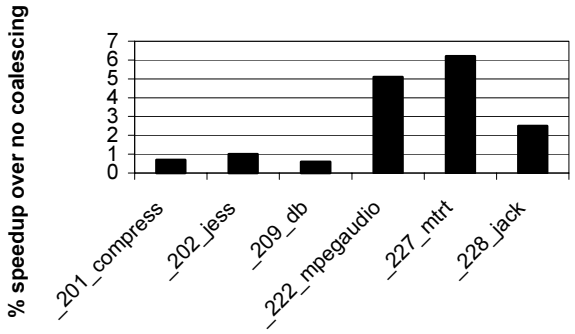

Figure 4. Register coalescing.

The complexity of the register coalescing algorithm is O(I + VB). The I component is because of the reverse traversal of the instructions. The VB component is because of the iteration over the live variables at each basic block boundary. This is the same asymptotic complexity that is required to build the live intervals; thus, this algorithm improves the results of linear scan register allocation without increasing its complexity. Figure 4 shows that register coalescing optimization improves performance by up to 6%. The best results are for the floating-point benchmarks because floating-point register moves have a high latency (5 cycles). The speedup for integer benchmarks is 1-2%.

4.2. Garbage collection support

The JIT compiler provides support for garbage collection at every instruction using a technique similar to

the one described in [20]. For each instruction, it computes the set of registers and stack locations that contain live references and interior pointers (pointers to the middle of the objects allocated on the heap) and records this information in a data structure called the GC map table. To enumerate the root set, the garbage collector iterates over the set of frames on each thread's runtime stack. For each frame, the garbage collector makes a callback into the JIT asking it to enumerate the set of live references for that frame and to unwind to the previous frame. The JIT compiler computes the set of live references for the frame using the GC map information.

The number of physical registers assigned to hold references and interior pointers affects the size of the GC map because of the GC map's encoding scheme. The compiler attempts to minimize the size of the GC map by attempting to minimize the number of physical registers that are assigned to hold references or pointers. The register allocator splits the integer physical registers into two groups: registers reserved for references and pointers, and registers that can contain values of any integral type. Register allocation is performed in two passes. In the first pass the allocator assigns registers only to references and interior pointers from an initial set of four available registers. If it runs out of registers, it does not add registers to the available set – rather it leaves the virtual registers unassigned. In the second pass, the allocator starts with a fresh set of four available registers, and assigns physical registers to all unassigned virtual registers, this time increasing the number of available registers if necessary. This technique allows the compiler to limit the number of registers used by references whenever possible without sacrificing the quality of register allocation.

5. Code scheduling

The basic unit of scheduling is an extended basic block (i.e., a linear sequence of instructions with a single entry point and multiple exits). The control flow exits from the middle of an extended block correspond to control flow caused by run-time exceptions. An extended basic block cannot span a function call. For each extended basic block the scheduler first builds a dependence graph whose nodes are instructions and whose edges correspond to dependencies. The graph contains an edge <I,J> if an instruction J is data or control dependent on instruction I. Each edge is annotated with the latency of the dependency.

The number of edges in the dependence graph affects the quality of the generated code as well as the time and memory required for scheduling. The method prolog sequence allocates a memory stack frame by adjusting the stack pointer and allocates a register frame using the `alloc` instruction; these instructions induce many dependencies in the program because any instruction that uses stack registers or the stack frame depends on them. The scheduler avoids these dependencies by assigning the maximum possible heuristic value to these prolog instructions, thereby guaranteeing that they are scheduled before the dependent instructions. This also allows the scheduler to schedule the prolog instructions in parallel with independent instructions.

The scheduler assigns instructions to execution cycles using cycle scheduling [11]**.** Given a choice between several ready instructions, the scheduler selects the one that has a maximum distance along any path to a leaf node of the graph. When several instructions have the same maximum distance the scheduler chooses the one that has the maximum number of successors. During cycle scheduling, the scheduler attempts to avoid over subscribing execution unit resources by using a resource vector to model each cycle's execution unit utilization [18]. The cycle scheduler, however, does not assign instructions to execution units – the subsequent template selection pass does this assignment. Template selection groups the instructions in each cycle into valid bundles, which effectively assigns instructions to execution units.

The reason for assigning execution units after cycle scheduling is that some instructions can be executed on more than one execution unit – assigning these instructions eagerly during scheduling can lead to sub-optimal code on Itanium. For example, an integer add can be executed on either the memory unit or the integer unit. When inserting an integer add instruction into a cycle, the scheduler records that the instruction may use either the integer unit or the memory unit, but it does not choose which one. The scheduler uses the generic machine resource approach [1] to guarantee that an execution unit will be available for the integer add instruction. Address computation instructions, however, are assigned to execution units during scheduling because this assignment affects the latency of the address computation instruction; Section 5.3 describes this in more detail.

5.1. Type-based memory disambiguation

Memory disambiguation is very important in a compiler for a statically scheduled machine such as Itanium because load operations tend to be on the critical path and false memory dependencies hinder instruction-level parallelism. Traditional compilers use potentially expensive analyses to determine whether two memory operations may access the same memory location. The JIT compiler uses the Java virtual machine's metadata to disambiguate memory accesses (i.e., it uses type-based memory disambiguation [9]).

The JIT compiler annotates each memory operand with the type and kind of the memory location it references. The

compiler derives type information from the Java byte codes and preserves this information throughout optimizations. The memory location kind is one of: object field, static field, array element, array length, stack argument, virtual table address, constant value, method address, spilled variable, callee-saved register, switch table element, and constant string address. The compiler further disambiguates field references using unique field identifiers provided by the virtual machine; for example, it can determine that the store instructions generated from the Java byte codes putfield #10 and putfield #15 refer to different memory locations even if both fields have the same type. Figure 5 shows that type-based memory disambiguation yields up to 2% performance improvement over a conservative approach that assumes that all memory accesses are aliased.

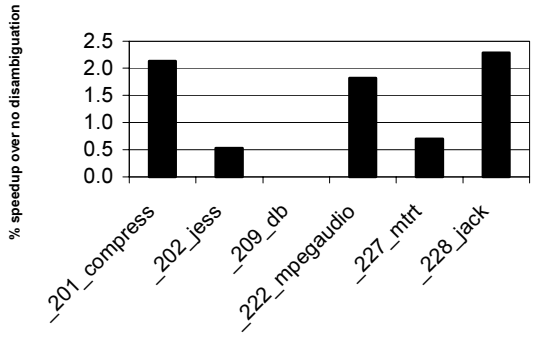

Figure 5. Type-based memory disambiguation.

5.2. Exception dependencies

The Java virtual machine [12][17] specifies that all object accesses must be checked at runtime – an attempt to access a field or method using a null object reference or an attempt to use an array index that is out of bounds causes an exception. Moreover, Java exceptions are precise – all visible side effects from instructions before the exception must appear to have taken place and no side effects from instructions after the exception may appear to have taken place [12].

The JIT compiler implements run-time exceptions explicitly by generating test-and-branch instructions that check the exception condition and transfer control to code that throws an exception if the check fails. These explicit checks introduce branches inside common code paths effectively creating extended basic blocks.

To enforce the precise exception semantics of Java the scheduler creates a dependence edge from an exception check branch to a subsequent instruction inside the extended basic block if that instruction is a memory store, writes a register that is live at the exception handler, is another exception check branch for a different exception type, or is a memory reference that may be guarded by the check (e.g., an instruction that loads a field of an object cannot be executed before the instruction that checks that the object reference is not null unless the compiler uses speculation). This strategy minimizes the number of dependence edges and allows load instructions that do not throw exceptions to be hoisted above exception check branches.

Figure 6 shows an example of a program fragment with a null pointer exception check. Assume there is no exception handler for the null pointer exception or such a handler does not use r17, r21, and f8. The scheduler generates only the dependency from the branch (instruction 2) to the object field load (instruction 4). Instructions 3 and 5 do not depend on the branch because their destination registers are dead on the branch taken path, and instruction 6 does not depend on the branch because it loads a static field (which cannot throw an exception) and because its destination register f8 is dead on the branch taken path.

```
1       (p6,p0) = cmp.eq r16, 0
2   (p6) br throw null pointer exception
3       r17 = add r16, 8
4       r18 = ld [r17]     // load object field
5       r21 = movl 0x000f14e32019000
6       f8 = fld [r21] // load static
```

Figure 6. Program fragment with exception dependencies.

The chart in Figure 7 shows the performance improvement from using extended block scheduling with precise dependencies between exception branches and load instructions over basic block scheduling. The chart in this figure shows that these techniques improve performance significantly; for example, the performance of 222_mpegaudio improves by more than 20%.

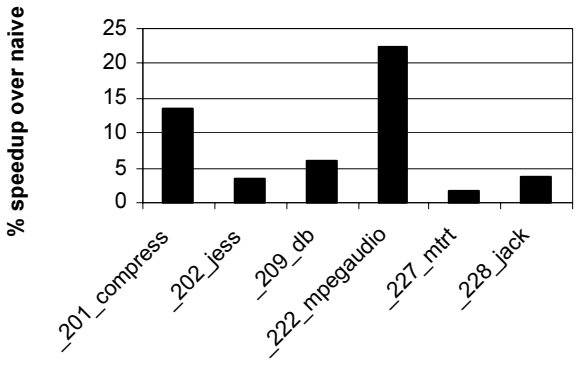

Figure 7. Extended basic block scheduling.

5.3. Modeling bypass latencies

The scheduler can model most bypass latencies easily because most instructions can only be mapped to a single execution unit type. An important exception is the latency from an address computation instruction to a load instruction that uses the address. An address computation can be executed in a single cycle on either an integer or a memory unit. There is no bypass latency to a load instruction if its address computation is executed on a memory unit and a bypass latency of one cycle if it is executed on an integer unit. As a result, the scheduler cannot model the true latency of the address computation instruction if it does not assign an execution unit to the address computation during scheduling.

Because memory address computation instructions tend to be on the critical path, the scheduler attempts to assign an address computation instruction to a memory unit during scheduling. If all the memory units for the cycle are already assigned the scheduler assigns the address computation to an integer unit and increments its dependence latency.

The chart in Figure 8 compares the performance improvement from accurately modeling the latency of address computation instructions over assuming there is no additional latency from an address computation to a load. This chart shows that this technique improves performance by up to 3%.

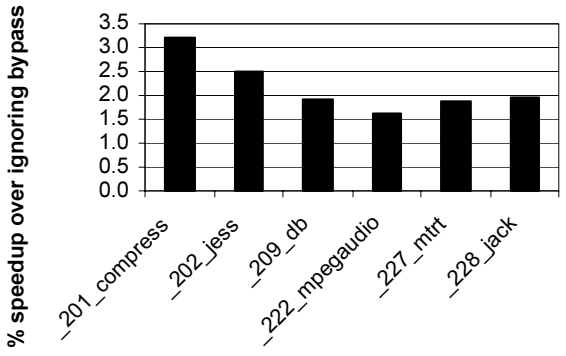

Figure 8. Modeling address computation bypass latency.

5.4. Template selection

The template selection phase groups instructions into bundles. It chooses an instruction slot for each instruction, fills empty slots with NOP instructions, and selects the templates that specify the execution unit types and cycle breaks.

Ignoring the execution unit and decode resources of the micro architecture during template selection can lead to over-subscription of processor resources -- for example, by assigning NOP instructions to units already assigned to other instructions or by using more than two templates for an instruction group (the Itanium processor decodes two bundles per cycle). Previous work [16] suggested using integer programming to find the optimal template assignment. Such an approach is too expensive in a JIT compiler. Instead, we designed a fast heuristics-based algorithm that performs template selection in linear time to the number of instructions and generates a contention-free schedule for most of the instruction cycles.

For each cycle in the schedule, the template selector iterates over the instructions in the cycle and greedily assigns each instruction to the first available instruction slot – that is, to the first empty slot that can contain an instruction of the given type under the restrictions imposed by the templates. Each cycle ends with a stop bit that can occur either in the middle or at the end of the bundle. Once a bundle is full, the compiler selects a template that is compatible with the instructions and stop bits in the bundle.

The critical part of this greedy algorithm is the heuristic that determines the order in which the scheduler considers the instructions. The template selector considers the instructions in order of their types whenever permitted by intra-cycle dependencies. The particular order -- M, F, L, I, A, B -- corresponds to the order in which the execution units appear in valid templates. The A-type instructions are scheduled after M- and I-type instructions to avoid over subscribing memory and integer execution units.

The example in Figure 9 illustrates how considering the instructions in an arbitrary order may lead to sub-optimal code. The heuristics will cause the greedy algorithm to optimally group the instructions in Figure 9a into two bundles that can be executed during the same cycle, as shown in Figure 9b. (The ".mfi" template of the first bundle indicates that the first, second and third instructions of that bundle use the M, F, and I units, respectively. The ";;" at the end of the second bundle indicates the stop bit.) Note that floating-point instructions i4 and i5 are assigned to different bundles because there is no template with two F-unit resources.

By comparison, the greedy algorithm that considers the instructions in their original order generates the non-optimal code shown in Figure 9c. It groups instructions i1, i2, and i3 into the first bundle. Finally, it must assign the NOPs to the unused instruction slots, which in this case can only be M-unit NOPs. The resulting code executes in two cycles.

The compiler uses several additional heuristics to prevent slot assignments that cause resource contention. First, it imposes restrictions on assigning an M-unit instruction to a middle slot of the bundle, since there are only two memory units available, and all templates except the .bbb template have an M-unit instruction in the first slot.

	Instruction	Type	Functional unit
(i1)	add a = b,c	A	M-unit or I-unit
(i2)	ld d = [x]	M	M-unit
(i3)	sxt e = f	I	I-unit
(i4)	faddh = u,t	F	F-unit
(i5)	faddj = v,w	F	F-unit

a) Instructions assigned to a cycle by the scheduler

```
{.mfi   ld    d = [x]        {.mmi   add a = b,c
        faddh = u,t                   ld    d = [x]
   sxt  e = f                         sxt   e = f
}                             } // stall –no M-unit for
nop.m
{.mfb  add a = b,c           {.mfi    nop.m
       fadd j = v,w                   faddh = u,t
       nop.b                          nop.i
;;}                           }
                              {.mfi    nop.m
                                       faddj = v,w
                                       nop.i
                              ;;}
```

b) Optimal template choice.

c) Result of greedy template selection with no heuristics.

Figure 9. Template selection example.

Thus, the compiler does not assign an M-unit instruction to the second slot unless the remaining instructions can be packed without using a second bundle with an M-unit first slot. Second, the compiler assigns A-unit instructions only to M-slots (I-slots) if the cycle under consideration uses all available I-units (M-units). Finally, the compiler chooses the execution units for the NOP operations that are least likely to cause the resource contention – namely, M-unit for the NOP in the first slot of the bundle, F-unit for the NOP in the second slot and B-unit for the NOP in the third slot.

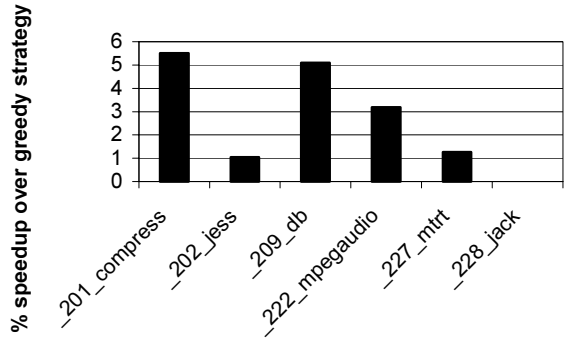

Figure 10. Template selection using heuristics.

The chart in Figure 10 compares the performance of the heuristic-based greedy algorithm with the naïve greedy algorithm not guided by the heuristics. Using the heuristics improves the performance by up to 5%.

6. Other optimizations

In this section we describe two potentially promising optimizations that we found to have only minor benefits: predication and sign extension elimination. We also describe a potential pitfall in Itanium code generation: Ignoring branch hint bits disables the hardware branch predictor, which hurts performance significantly (up to almost 50% on one benchmark) .

6.1. Predication

Predication (also known as if-conversion [3]) is a program transformation that removes a branch by predicating the instructions that are control dependent on the branch. A simple example of predicated code is shown in Figure 11.

```
     cmp.lt (p6,p7) = r17, 0      cmp.lt  (p6,p7) = r17, 0
(p6) br L1                   (p6) mov    r8 = 1
     mov   r8 = 1            (p7) mov    r8 = 0
     br L2                        br.ret
L1:  mov   r8 = 0
L2:  br.ret
```

a) Unpredicated code

b) Predicated code

Figure 11. Example of predication.

Predication can have both a positive and negative affect on execution time: It can speed up execution by reducing stalls caused by mispredicted branches, but can also slow down execution by increasing the number of executed instructions In general, predication is profitable only if the reduction in branch misprediction stall cycles is greater than the increase in the number of wasted execution cycles that are a result of squashed instructions. A compiler must accurately model resources and consider profile information to guarantee performance improvement from predication.

Our compiler uses a simple version of predication. Namely, it predicates simple hammocks (control flow structures generated from simple if-then-else statements) that are balanced (i.e., have the same number of instructions in the then and else blocks). It also avoids predicating hammocks that contain calls: The Itanium processor implements calls as branches; therefore, replacing a potentially mispredicted branch with one or more mispredicted calls will not result in a performance improvement. The chart in Figure 12 shows that this simple

predication technique improves performance by up to 2% but can also degrade performance by 1%. These results are consistent with the finding about general predication in [7].

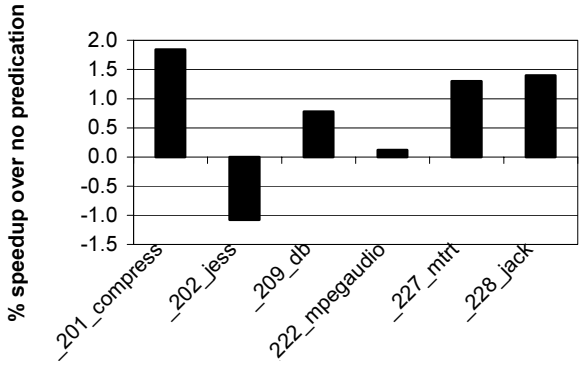

Figure 12. Predication.

6.2. Using branch hints

The first version of the compiler naïvely ignored the branch hint bits and used the default value of zero, which happened to be statically taken. As a result, the compiler did not use the hardware branch prediction mechanism at all. In a later version, the compiler specified unconditional branches to be statically taken and specified all other branches to be dynamically not taken. This simple strategy worked amazingly well as shown in Figure 13.

We tried to further refine branch hints by specifying exception branches as statically not taken and loop backward branches as dynamically taken; however, this refinement did not have a measurable affect on performance.

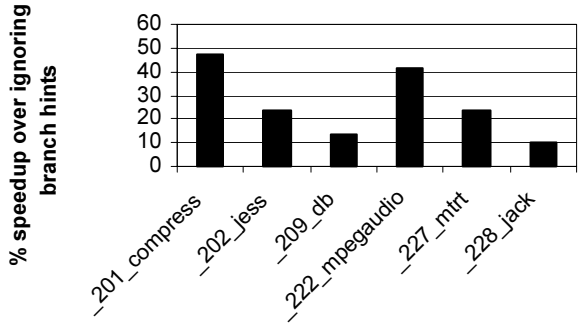

Figure 13. Using branch hints.

The compiler generates hints for indirect branches in the move to branch register instruction. These hints, however, do not have a measurable affect on performance because they are effective only if the move instruction precedes the branch instructions by at least 9 cycles. The scheduler (which is not as aggressive as a global code scheduler in a static ILP compiler) cannot find enough instruction-level parallelism to take advantage of indirect branch hints.

6.3. Eliminating sign extension

One potential optimization we looked into is sign-extension elimination. There are two simple ways to do this: eagerly and lazily. An eager approach generates sign extension after every definition of a 32-bit number that might generate an incorrectly sign-extended 64-bit number. This is very expensive, as any arithmetic operation can generate incorrect sign extension in case of over or underflow. Another approach is to generate sign extension before every use of a 32-bit number that requires it. This is the approach we implemented in our compiler. The most frequent case of an instruction requiring a sign-extended operand is array element address computation. Most of the indices are 32-bit integer numbers that have to be sign-extended before they can be scaled and added to the 64-bit array address.

There are many opportunities for eliminating sign extension. Yet, Itanium is a multi-issue processor, and eliminating sign extension will result in the performance improvement only if the sign extension happens to be on a critical path.

We measured the upper bound on the potential performance improvement from sign-extension elimination by unsafely disabling sign-extension generation. Most of the benchmark programs executed correctly and the performance improvement was no more than 1%. This is because most of the sign extension was for array indices, which were not on the critical path (they can be done in parallel with computing the address of the first array element). We conclude that sign-extension elimination is not an important optimization if the compiler does not perform aggressive global optimizations.

7. Conclusions

The Itanium processor relies heavily on ILP-extracting compiler optimizations for performance. Implementing a JIT compiler for Itanium is challenging because ILP-extracting optimizations tend to be expensive and take time to implement. The code generation techniques described in this paper are lightweight and yield efficient Itanium code. The techniques rely on heuristics to model the Itanium micro architecture and on JVM semantics to extract ILP. Our measurements show that the goal of inexpensive but effective optimizations on Itanium is achievable using the techniques described in this paper.

References

[1] A. Adl-Tabatabai, T. Gross, G.Y. Lueh and J. Reinders. Modeling Instruction-Level Parallelism for Software Pipelining. *IFIP WG10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, North Holland, January 1993, pp. 321-330.

[2] V. Aho, R. Sethi, and J. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, second edition, 1986.

[3] J. R. Allen, K. Kennedy, C. Portefield, and J. Warren. Conversion of Control Dependence to Data Dependence. *Symposium on Principles of Programming Language,* January 1983, pp. 177-189.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive Optimizations in the Jalapeno JVM. *Conference on Object-Oriented Programming, Systems, Languages & Applications*, October 2000, pp. 47-64.

[5] G.J. Chaitin. Register allocation and spilling via graph coloring. In Proceeding of the ACM SIGPLAN 1982 *Symposium on Compiler Construction*. pp. 201-207.

[6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation via Coloring. *Computer Languages,* Vol. 6, No. 1, 1981, pp. 47-57

[7] Y. Choi, A. Knies, L. Gerke, and T. Ngai "The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel Itanium Processor". *34th Symbposium on Microarchitecture,* Austin, TX, December 2001, pp. 182-191.

[8] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. *Conference on Programming Language Design and Implementation,* October 2000, pp. 13-26.

[9] A. Diwan, K. S. McKinley, and J.E.B. Moss. Type-based Alias Analysis. *Conference on Programming Language Design and implementation,* May 1998, pp. 106-117.

[10] Free Software Foundation. GNU Classpath. Availabe at http://www.gnu.org/software/classpath

[11] P. B. Gibbons and S. S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. *Symposium on Compiler Construction,* July 1986.

[12] J. Gosling, B. Joy and G. Steele. The Java Language Specification. Addison-Wesley, 1996.

[13] Intel Corporation. Intel® Itanium® Architecture Software Developer's Manual. Available at http://developer.intel.com/design/itanium/manuals.

[14] Intel Corporation. Intel® Itanium® Processor Reference Manual for Software Optimization. Available at http://developer.intel.com/design/itanium/manuals

[15] Intel Corporation. Open Runtime Platform (ORP). Available at http://orp/sourceforge.ne

[16] D. Kaestner and S. Winkel. ILP-based Instruction Scheduling for IA-64. *Workshop on Languages, Compilers, and Tools for Embedded Systems,* June 2001, pp. 145-154.

[17] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Second Edition. Addison-Wesley, 1999.

[18] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[19] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 5, September 1999, pages 895-913.

[20] J.M. Stichnoth, G.-Y. Lueh, and M. Cierniak. Support for Garbage Collection at Every Instruction in a Java Compiler. *Conference on Programming Language Design and Implementation*, May 1999, pp. 118-127.

[21] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Available at http://www.spec.org/osg/jvm98.

[22] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. *Conference on Object-Oriented Programming, Systems, Languages & Applications*, October 2001, pp. 180-194

[23] Sun Microsystems. The Java hotspot Virtual Machine. White paper available at http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html